Chirp: A Practical Global Filesystem for Cluster and Grid Computing

Douglas Thain, Christopher Moretti, and Jeffrey Hemmes
Department of Computer Science and Engineering, University of Notre Dame

Abstract. Traditional distributed filesystem technologies designed for local and campus area networks do not adapt well to wide area grid computing environments. To address this problem, we have designed the Chirp distributed filesystem, which is designed from the ground up to meet the needs of grid computing. Chirp is easily deployed without special privileges, provides strong and flexible security mechanisms, tunable consistency semantics, and clustering to increase capacity and throughput. We demonstrate that many of these features also provide order-of-magnitude performance increases over wide area networks. We describe three applications in bioinformatics, biometrics, and gamma ray physics that each employ Chirp to attack large scale data intensive problems.

Keywords: filesystem, grid computing, cluster computing

1. Introduction

Large scale computing grids give ordinary users access to enormous computing power at the touch of a button. Production systems such as the TeraGrid[6], the Open Science Grid[7], and EGEE[1] all regularly provide tens of thousands of CPUs to cycle-hungry researchers in a wide variety of domains. These and similar systems are most effective at running CPU-intensive jobs with small amounts of input and output data.

Data-intensive jobs are not as easy to run in a computational grid. In most systems, the user must specify in advance the precise set of files to be used by a grid job. For many applications composed of multi-process scripts, interpreted languages, and dynamically linked libraries, determining this list may be very difficult for the end user to compose. In some cases it may simply be impossible: in a complex application, the set of files to be accessed may be determined by the program at runtime, rather than given as command line arguments. In other cases, the user may wish to delay the assignment of data items to batch jobs

[0] **This is a preprint manuscript of:** Douglas Thain, Christopher Moretti, and Jeffrey Hemmes, "Chirp: A Practical Global File System for Cluster and Grid Computing", *Journal of Grid Computing*, Springer, accepted for publication in 2008. DOI: 10.1007/s10723-008-9100-5

until the moment of execution, so as to better schedule the processing of data units.

For these kinds of data-intensive applications, a distributed filesystem could provide the familiar form of run-time access using the same namespace and semantics found on a local machine. However, despite many decades of research in distributed filesystems, none are well suited for deployment on a computational grid. Even those filesystems such as AFS [29] designed to be "global" are not appropriate for use in grid computing systems, because they cannot be deployed without intervention by the administrator at both client and server, and do not provide consistency semantics or security models needed by grid applications.

To address this problem, we have designed the Chirp distributed filesystem for cluster and grid computing. Chirp allows an ordinary user to easily deploy, configure, and harness distributed storage without requiring any kernel changes, special privileges, or attention from the system administrator at either client or server. This important property allows an end user to rapidly deploy Chirp into an existing grid (or several grids simultaneously) and use it to access data transparently and securely from multiple sources.

In this paper, we provide a broad overview of all aspects of the Chirp distributed filesystem, building on several previous publications [44, 27, 43, 32, 47] that introduced different technical elements independently. Section 2 discusses the unique properties needed for a grid filesystem. Section 3 reviews the related work. Section 4 gives an overview of the components of Chirp. Section 5 describes the three available namespaces for file access. Section 6 describes the flexible security model, including four authentication types, access control lists, and distributed authorization. Section 7 introduces several new system calls that improve usability and wide area performance. Section 8 describes three models for consistency semantics: strict, snapshot, and session semantics. Section 9 describes how Chirp servers can be clustered into larger structures. Section 10 shows how Chirp relates to existing grid middleware. Finally, section 11 describes three applications of the filesystem to bioinformatics, biometrics, and gamma ray physics.

2. Desired Properties

Why not make use of an existing file or storage system for the purpose of grid computing? To answer this question, we first note that users of grid computing systems have the following unusual needs which are not met by existing designs.

Rapid Unprivileged Deployment. A user that submits jobs to a wide area grid computing system will find them running on a a completely unpredictable set of machines, perhaps under a temporary user account, without any opportunity to prepare the machine or account for use by the job. To live in this environment, the capability to access remote storage must be brought along with the job itself, without relying on any help from the local operating system or administrator. A filesystem that relies on the client modifying or updating the local kernel in any way cannot be used in such an environment.

Likewise, many users wish to export access to data from locations in which they have no special privileges. A researcher at a university may have access to data on a storage appliance from a workstation, but no special privileges on either the appliance or the workstation. That user should still be able to export his or her data (within certain security constraints) to jobs running on a computational grid. Or, perhaps a user wishes to stage data into the head node of a remote cluster in order to achieve better performance through data locality. If a file server can be easily deployed on the remote head node, it can be used to both accept incoming data over the wide area network as well as serve it to jobs on the local network.

Support for Unmodified Applications. Some users may be able to modify their applications to use a grid file server, but they must be both highly motivated and technically sophisticated. Even in this case, changing an application is only practical if it consists of a small amount of source code running in a single process. For most users, changing an application in order to use the grid is highly undesirable. For commercial applications, it may be impossible.

An ideal grid filesystem would allow the execution of unmodified binary applications. In particular, it should support non-trivial programs, particularly scripting languages that invoke multiple processes, load dynamic libraries, and involve complex relationships between programs. Such complex applications are often developed in the easy environment of a local workstation, and then deployed to a computational grid in their full complexity.

Support for Both Large and Small I/O. Most grid I/O systems focus on one of two distinct modes. File transfer systems such as GridFTP [10] provide high-bandwidth movement of large files from place to place, while file access systems such as GridNFS [28] provide small file access to data already in place. The former is suitable for data staging, while the latter is more suitable for on-demand access in the local area.

Although it is possible to deploy both types of servers simultaneously on the same system, such systems are certain to have different semantics and access control models, leading to deployment complexity, confusion for the user, and unusual semantics. For example, what happens if one server is caching some data that the other is unaware of? Ideally, a grid filesystem should provide both large file transfer and small file access within the same framework of access control and consistency semantics.

Flexible Security Policies Traditional filesystem controls do not make it easy for users to specify how data is shared outside of the local machine. Traditional Unix is the worst, allowing each file to be associated with exactly one user and one group defined by the administrator. Even in systems that implement access control lists, the members of the list can only be locally-defined users and groups.

Users of grid computing systems need flexible access control mechanisms that allow them to share data both with local users and with remote users, or perhaps their own remote jobs, identified by cryptographic credentials. In addition, when collaborating with other users in their virtual organization [23], they need to define and refer to groups of distributed users from many different sites simultanously. To this end, a grid filesystem should provide a highly flexible access control mechanism.

Tunable Performance Tradeoffs Traditionally, filesystems designers have chosen fixed consistency semantics suitable for an assumed client workload. Well known examples include lazy semantics in NFS [37], precise Unix semantics in Spritely NFS [39], and open-close snapshot semantics in AFS [29]. In general, by giving up timeliness, performance and availability can be improved.

However, no single set of consistency semantics is likely to satisfy all users of a grid filesystem. Deterministic batch jobs may be quite happy with very lazy updates, allowing for hours or days to pass between up-to-date checks, because it is known in advance that the input executables and data will not be changed. On the other hand, interactive grid jobs that are steered by an external users may wish to check for changed inputs at every opportunity. We cannot choose a single set of consistency semantics, but must allow each user to make a different tradeoff between timeliness, availability, and performance.

3. Related Work

With the above requirements in mind, we may review grid storage systems and make the case that a new grid filesystem is needed.

The most widely used tool for large data transfer in grid computing is GridFTP [10], which provides authentication via the Grid Security Infrastructure (GSI) [22] and high bandwidth data transfer through parallel streams. In addition, client libraries such as GASS [18] and XIO [9] provide simple interfaces that allow slightly-modified applications to stream data to and from remote servers, with various transformations such as caching, compression, and retry easily specified.

Although GridFTP is an excellent *data transfer system*: it was never designed to be a *file system*. Using a local cache, it is technically possible to connect FTP servers to a filesystem interface. For example, UFO [8], SlashGrid [5], and Parrot [46] all allow the user to read and write remote FTP files, but the results are often unexpected because FTP does not provide the precise semantics required by standard applications. For example, many FTP servers do not distinguish between a non-existent directory and a directory with no files: both result in the same error. Across FTP servers, there is no consistent way to retrieve file metadata such as modification time and owner. Many FTP servers consistently return error code 550 on any kind of failure without indicating further details. Such vagaries might be overlooked by an interactive user, but cause havoc in real applications that depend on the subtle distinctions between errors such as *file not found*, *access denied*, and *not a directory*.

Grid storage systems such as SRM [38], IBP [35, 15], and NeST [16] have focused on problem of managing limited storage space shared by multiple users. Freeloader [49] provides a high performance file object cache on a cluster of commodity machines. These systems all provide some form of file input and output, but do not have the intention to provide a Unix-compatible interface.

Other systems have targeted the filesystem interface directly. LegionFS [26], Ceph [51], L-Store [4], GFarm [42] all provide a filesystem constructed from multiple custom storage devices. However, because these each employ custom underlying storage devices, they are not suitable for exposing *existing* filesystems to the wide area. GPFS-WAN [12] provides access to several supercomputer centers from nodes on the NSF TeraGrid. Using custom protocols and specialized networking and storage hardware, it can provide very high performance within that context. However, because of the kernel-level client implementation and specialized hardware, it cannot be easily accessed or deployed outside of the TeraGrid. SRB [13] provides access to large datasets indexed by complex metadata. It is appropriate for the curation of multi-TB archival datasets, but it is a heavyweight system not designed to be rapidly deployed by a single user.

A variety of systems have augmented NFS [37] for use on the grid. PUNCH [20] provides on-demand configuration and access to multiple NFS servers across a campus. VegaFS [31] adds PKI authentication. FT-NFS [14] adds intra-cluster capacity sharing. GridNFS [28] adds

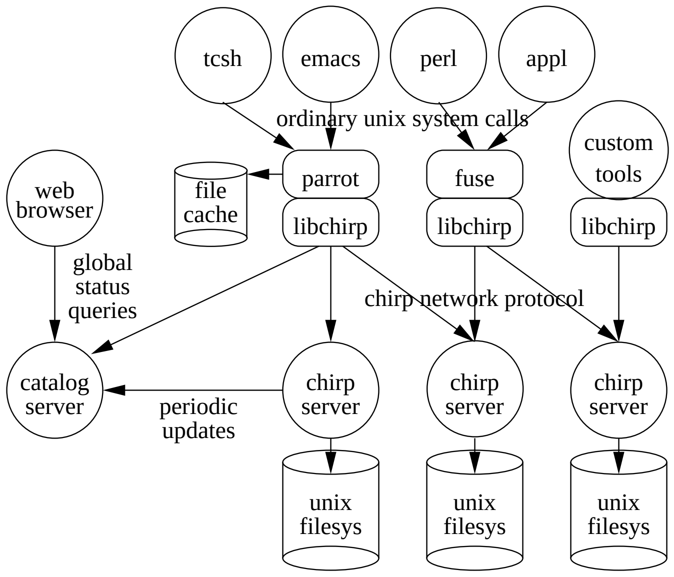


Figure 1. Overview of the Chirp Filesystem

GSI security and clustering capabilities. WOW [24] employs NFS over wide area virtual private networks. Although based on familiar technology, NFS variants are not well suited to grid computing. Primarily, NFS requires a kernel level implementation for access to inode numbers at the server, and administrator assistance to mount the filesystem at the client. In addition, the block-based nature of NFS makes it difficult to achieve high bandwidth over the wide area.

Other grid I/O systems provide a transparent bridge from the user's submitting machine to the site of an executing job. These systems typically provide less flexibility than a full filesystem, but can be be deployed with little or no user intervention. Examples of these systems include the Condor remote system call library [48], Bypass [45], XUnion [50], and PDIO [41]. Such systems create convenient private data spaces to be accessed by a single user, but do not provide the full generality of a multi-user filesystem.

At the time of writing, the Global Grid Forum [3] has two working groups creating standards for a grid file system and a POSIX-like I/O interface using the Open Grid Services Architecture [21].

4. Overview

Figure 1 shows the main components of Chirp. A *chirp server* is a user-level process that runs as an unprivileged user and exports an existing local Unix filesystem. A system may have multiple servers, each of which periodically sends a status update listing address, owner, available space, and similar details to a well-known *catalog server* via UDP. The catalog server can be queried via HTTP to obtain a global list of available servers. If redundancy is desired, each server can be configured to report to multiple catalogs on different hosts.

Clients may access Chirp file servers in several ways. A library *libchirp* is provided that allows clients to program directly to the Chirp protocol, although we expect that few (if any) applications will wish to do so. Instead, applications typically connect to the filesystem through one of two adapters – Parrot or FUSE – which present the entire space of Chirp servers as an ordinary filesystem. A few additional custom tools are provided to manipulate access controls, allocations, and other features that do not map directly to Unix operations.

The network protocol is based on TCP. On the first remote file request to a server, *libchirp* makes a TCP connection to that server, authenticates, and then issues the file request. A client may hold open connections to multiple servers, and a server may have connections to multiple clients. Network failures that cause the TCP connection to hang or fail are transparently repaired by *libchirp*. Servers regularly drop connections that are idle for one minute, in order to limit unused kernel state, with the side effect of regularly exercising the recovery code. A single connection is used for both control and data, thus optimizing access to small files, while allowing for large TCP windows to be opened and maintained on high bandwidth network connections.

Chirp is easy to deploy. A single Chirp server is started as follows:

```
% chirp_server -r /data/to/export
```

By default, data exported by this simple command will only be accessible to the invoking user, until the access controls are modified, as discussed below. Alternate catalog servers may be deployed in a similar manner. Because each catalog server maintains state only in memory, it does not require the installation of a database or other supporting software.

As noted above, Chirp simply stores files and directories in the ordinary way in the underlying filesystem. (With the exception of access control lists, described below.) The advantage of this approach is that existing data may be exported without moving or converting it. The drawback is that the capacity and performance of a Chirp server is constrained by the properties of the underlying kernel-level filesystem.

There are several common ways of using Chirp:

– **Personal File Bridge.** Because Chirp simply makes use of the underlying Unix filesystem interface, it can be used to securely export many existing storage systems to the grid. Suppose that a potential grid user has important data stored in a campus AFS cell and wishes to access it from a foreign grid without AFS installed. The user can easily start a Chirp server under his or her own AFS credentials, and set the access controls to allow access to his or her grid credentials. Then, jobs running in the grid can access AFS through the Chirp server.

- **Shared Grid File Server.** Chirp has a much more expressive access control system than a traditional Unix filesystem. Users may define their own groups and access control lists that refer to a mix of hostnames, local users, grid credentials, and Kerberos users, even if those subjects do not have accounts on the local machine. In this way, Chirp is a natural way to share data with a virtual organization of users spread across multiple institutions: each user can access the server as a first-class citizen, without requiring the local administrator to create local accounts.
- **Cluster File System.** A collection of Chirp servers, whether personal or shared, are easily deployed on a collection of nodes in a computing cluster, and then joined into a common namespace, both to provide additional I/O bandwidth as well as the aggregate capacity of a cluster. Users may reconfigure subsets of this cluster for various tasks, such as online analysis, remote fileservice, or distributed backup and recovery. We have experience operating such a cluster of 250 machines and 40TB of storage since 2005.

So far, we have given a very high level overview of Chirp. In each of the following sections, we will describe different aspects of the system in greater detail.

5. Interface

Users have several choices for interfacing to Chirp servers: the Chirp library, the Parrot agent, and the FUSE toolkit. Each offers a different tradeoff in performance, functionality, and deployability.

The most direct way to access Chirp is through the *libchirp* library, which presents an interface similar to the Unix I/O interface. Following is an (incomplete) selection of calls in the library:

```
chirp_open   ( path, flags, mode, timeout )
chirp_pread  ( file, buffer, length, offset, timeout )
chirp_pwrite ( file, buffer, length, offset, timeout )
```

```
chirp_fstat  ( file, statbuf, timeout )
chirp_close  ( file, timeout )

chirp_stat   ( path, statbuf, timeout )
chirp_unlink ( path, timeout )
chirp_mkdir  ( path, timeout )
chirp_rmdir  ( path, timeout )
chirp_getdir ( path, timeout )
```

In addition, *libchirp* also has several calls that invoke higher order functions not found in Unix. The following calls get, put, and move entire files and manage identity and access controls.

```
chirp_getfile  ( path, localfile, timeout )
chirp_putfile  ( path, localfile, length, mode, timeout )
chirp_thirdput ( path, targethost, targetpath, timeout )

chirp_whoami   ( path, timeout )
chirp_setacl   ( path, subject, rights, timeout )
chirp_getacl   ( path, timeout )
```

Applications can be modified to access *libchirp*, either manually or by textual substitution. This provides the fastest possible access to remote data. Several command-line tools are provided that invoke *libchirp* directly to transfer files and perform other managmenet tasks. However, most users access Chirp through a more convenient interface.

Parrot [46] is the tool most commonly used to access Chirp. It is an *interposition agent* [30] that invokes ordinary programs, traps all of their system calls through the debugging interface, and modifies the actions and the results of those that refer to remote files, while leaving local files untouched. For example, to processes running on top of Parrot, the Chirp filesystem is visible under the path `/chirp`, while GridFTP servers are available under the path `/gridftp`. Parrot is an ordinary program that may be carried along with a batch job. It can be used in a normal shell environment as follows:

```
% parrot tcsh
% cd /chirp
% ls
alpha.nd.edu
beta.nd.edu
gamma.nd.edu
...
```

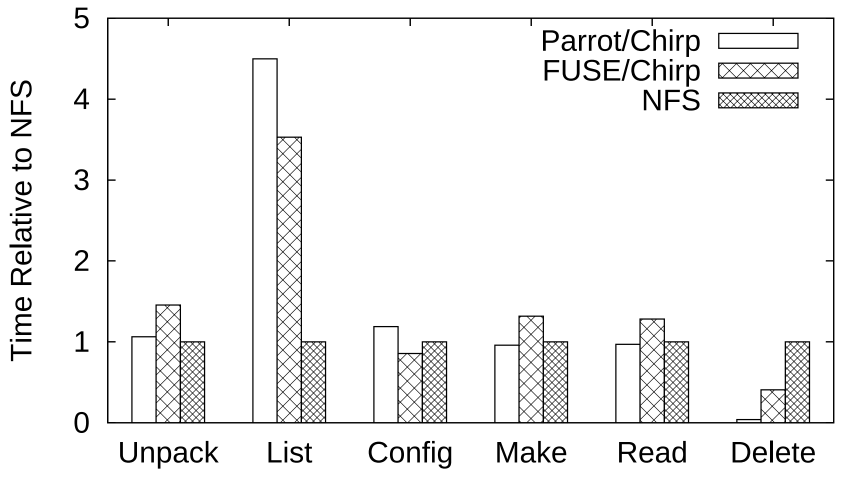


Figure 2. Performance Comparison of Chirp Interfaces against NFS

An alternative to using Parrot is FUSE [2], which is commonly used to implement user-level filesystems. FUSE has two components: a generic kernel-level module, and a filesystem-specific user-level module. Once the administrator has installed the kernel module, then any unprivileged user can mount Chirp into any directory in which he or she already has write access. Multiple users on the same machine may mount Chirp in different parts of the filesystem simultaneously. The FUSE kernel module will become a standard part of future Linux kernels, so this may be a more practical way of employing Chirp in the future. FUSE can be used in a normal shell environment as follows:

```
% mkdir /home/betty/chirp
% chirp_fuse /home/betty/chirp
% cd /home/betty/chirp
% ls
alpha.nd.edu
beta.nd.edu
gamma.nd.edu
...
```

Which interface should the user employ? The choice does not have a significant effect on performance. To demonstrate this, we ran an Andrew-like benchmark comparing Parrot/Chirp, FUSE/Chirp, and NFS all against the same file server across a 100Mbit campus-area network. In the benchmark, a copy of the Chirp software package is stored on a file server, and manipulated by a remote client. The client copies the package, unpacks it, lists every directory, builds the software, reads all the created files, and then deletes the directory. (Consistency semantics will be discussed in detail below, but in this case, we choose file snapshot semantics for Chirp.)

Figure 2 shows the runtime of each stage, relative to the performance of NFS. Both Parrot and FUSE are significantly slower than NFS for listing files, because listing involves a large number of system calls, each of which pays a similar penalty through both Parrot and FUSE. On the delete stage, both are faster than NFS, because the Chirp protocol allows a recursive delete to be implemented with a single RPC. In the remaining stages, both Parrot and FUSE are within twenty percent of NFS performance, but neither with a clear benefit.

So, the choice of interface should be made on more subjective grounds. FUSE is more portable than Parrot: it is implemented on many variants of Linux, on Apple OSX, and ports are in progress to other Unix variants. Parrot currently only runs on IA32 and AMD64 Linux. On the other hand, Parrot does not require root privileges to install, and allows for more flexibility in the namespace, as we describe below. FUSE restricts the user to attaching a new filesystem to an empty directory such as `/home/betty/chirp`. Parrot also allows the user to take advantage of custom system calls that improve the performance and usability of Chirp.

In order to describe the full functionality of Chirp, we will assume that the user is employing Parrot for the remainder of this paper.

6. Namespace

Chirp filesystems can be accessed through three distinct namespaces, each shown in Figure 3: the absolute namespace, the private namespace, and shared namespaces. The user can select what namespace to use at runtime with command line arguments.

The **absolute namespace** is found in the path `/chirp` and represents all of the known Chirp servers as a directory with the hostname and port. Under each directory, the user may manipulate the contents of each server independently. This namespace allows the user or administrator to manage data when the physical location is important for performance or semantic reasons. For example, a user may copy data from one server to another with the following command:

```
cp /chirp/c05.nd.edu/data /chirp/desktop.nd.edu/incoming/
```

While an administrator can inventory the free disk status of all known servers like this:

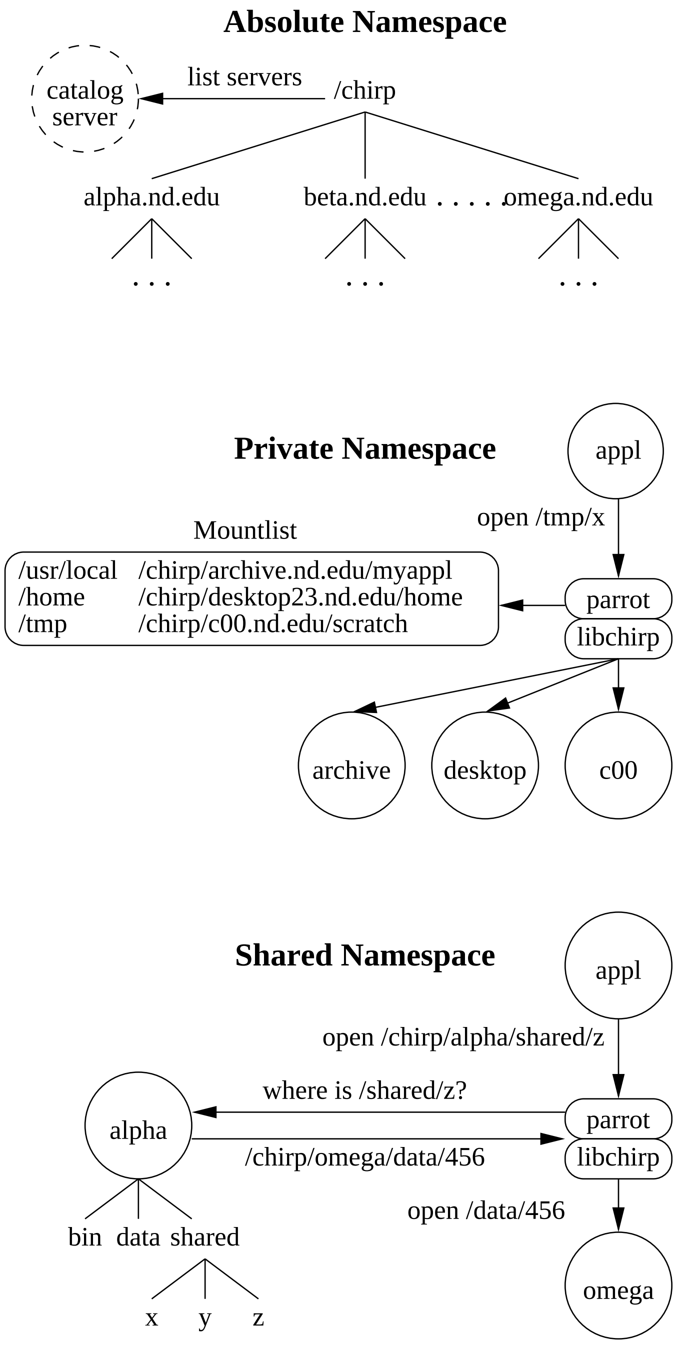


Figure 3. Three Namespaces of the Chirp Filesystem

```
df /chirp/*
```

The absolute namespace is implemented with the assistance of the catalog server. Each running server periodically sends a UDP update to the catalog server, listing its vital statistics. Clients that access the absolute namespace (for example, by listing `/chirp`) send an HTTP request to the catalog server to retrieve the list of servers and their properties. To avoid introducing serious loads on the central server, this result is cached for several minutes, assuming that the set of file servers does not change rapidly. Operations on files and directories below the top level result in a direct connection to the relevant server.

AFS [29] has a similar top-level namespace that lists *cells* rather than individual file servers. A problem with the AFS namespace in practice is that accidental references to the top-level result – such as long directory listings or auto-complete – result in sequential queries to all known cells, which essentially brings a client to a halt. To avoid this problem, Chirp implements `stat` calls on individual directory names by simply consulting the cached catalog query result and filling in the size and time fields with the available space on the server and the time the server last updated the catalog. Thus, common operations on the top level are satisfied locally and quickly.

The **private namespace** can be constructed on a per-application basis by passing a *mountlist* to Parrot that lists how requests for logical file names should be mapped to physical locations. This allows jobs running in a grid to retain existing naming conventions, even while data is relocated as needed for performance or policy reasons. Here is an example mountlist that might be used by an application that loads software from a well-known archive, reads configuration info from the user's home desktop, and employs temporary space on the cluster head node:

```
/usr/local   /chirp/archive.nd.edu/myapps
/home        /chirp/desktop23.nd.edu/home
/tmp         /chirp/c00.cse.nd.edu/scratch
```

Parrot loads the mountlist into memory, consults it on every reference to a pathname, and rewrites paths as needed. The cost of these memory operations is very small compared to even local I/O operations, so the performance overhead is neglible.

The **shared namespaces** allow multiple users to construct collaborative collections of files that may be physically scattered across multiple servers. A shared namespace is rooted in a Chirp directory that is well-known to its participants. The directory contains nested directories and files, just like an ordinary filesystem, except that the file entries are instead pointers to files stored on other Chirp servers. The client libraries transparently traverse these pointers to create the illusion of a very large filesystem spread across multiple devices. Access controls on the shared namespace can be set so that collaborating users can read and modify the namespace as needed. The shared namespace is typically used for clustering servers, described in Section 10.

7. Security

Different users and applications within the same grid can have very different needs for security. Strong security mechanisms exact a price of increased user burden and decreased performance, so the mechanism must be chosen to be proportionate to the user's needs. For example, a user processing sensitive data on a grid might require expensive user authentication for each individual client, while another distributing software to a campus might be satisfied by simply identifying the hostname of the client.

Chirp offers mechanisms to satisfy a wide range of security needs within the same software framework. We separate out the concerns of authentication, which is the process of identifying users, from the concerns of authorization, which is the process of granting or denying access to a known user.

7.1. Authentication

Chirp implements several authentication methods. When a client and server connect, they negotiate a mutually-acceptable authentication method, and then attempt to authenticate. If the attempt fails, the client may propose another method and then try again. By default, *libchirp* attempts methods for which credentials are locally available, which satisfies most users, but the method may also be selected manually by the user. The following methods are available:

– **Kerberos** [40] is a widely-used distributed authentication system based on private key credentials. If the user has previously logged into their workstation via Kerberos, then the appropriate host ticket is obtained and passed to the Chirp server without any additional action by the user. This method yields a Chirp subject like `kerberos:betty@nd.edu`.

– **Globus** employs the Globus Grid Security Infrastructure [22], which is based on public key cryptography. The user must first generate a proxy certificate by invoking `grid-proxy-init`, and then may authenticate to multiple Chirp servers without any further steps. This method yields a long Chirp subject name like: `globus:/O=Notre Dame/OU=Computer Science/CN=Betty Smith`

- **Unix** employs a challenge-response in the local filesystem: the server challenges the client to touch a file like `/tmp/challenge.123`. If the client succeeds, the server infers that the client's identity is that of the owner of the challenge file. This method only serves to identify clients running on the same host, and is typically used to identify the owner of the server for administrative tasks. This method yields a Chirp subject like `unix:betty`.
- **Hostname** employs a reverse-DNS lookup to identify the calling user based on their hostname. This method is obviously not suitable for securely identifying a given user, but may be suitable for distributing data to an entire organization. This method yields a Chirp subject like `hostname:desktop23.nd.edu`.

The subject name generated by the authentication step is then used to perform authorization for each access to the filesystem. Note that even if a client holds multiple credentials, it may only employ one subject name at a time. To switch credentials, the client must disconnect and re-authenticate.

7.2. Authorization

Authorization is controlled by per-directory ACLs, much like AFS [29]. Each directory controlled by a Chirp server contains a hidden file `.__acl` that lists the subjects authorized to access that directory. Each ACL entry is simply a (possibly wildcarded) subject name followed by a string of characters listing the rights granted to that subject.

The available rights are: **R** to read files or their metadata, **W** to write and create files, **L** to list the directory, **D** to delete files or that directory, **X** to execute programs and **A** to modify the ACL on that directory. For example, the following ACL gives full access to Betty (when using Kerberos credentials), most access to a friend identified by Globus credentials, and read-only access to any other host in the hierarchy `nd.edu`:

```
kerberos:betty@nd.edu           RWLDA
globus:/O=NotreDame/CN=Friend  RWLD
hostname:*.nd.edu               RL
```

An additional access right is needed to handle the common case of a server that is used as a shared staging point for data. In this case, we do not want to simply give a large class of users read and write access to the root directory, because they could too easily interfere with each other's work. Instead, we wish to allow them to create a new directory, manipulate files within that directory, and share those files at their discretion.

To allow this, we introduce the **V** right, which only allows a user to create a new directory, to which will be given the set of rights attached to the **V** right, with wildcards resolved. We call this process *namespace reservation*, indicating that the user is setting aside a portion of the namespace for their own use. (Note that it is *not* space reservation, which is a different mechanism entirely.)

For example, suppose that a server has the following ACL, which allows a large class of users to invoke `mkdir` and nothing else:

```
hostname:*.cse.nd.edu          V(RWDL)
globus:/O=NotreDame/*          V(RWLDA)
```

Now, if `globus:/O=NotreDame/CN=Betty` issues `mkdir /mydata`, the new directory will be created with the following ACL:

```
globus:/O=NotreDame/CN=Betty   RWLDA
```

Betty can then manipulate files in that directory, and if desired, use the **A** right to give additional access to any other user that she wishes to collaborate with.

7.3. Distributed Authorization

The notion of *virtual organizations* [23] is an important driver of grid computing. Many users of grid computing consist of affiliated people, each working for a different institution, but working on a common research project that shares data, equipment, and other resources.

To manage a large group, it is impractical to list every member of the group individually in every single ACL spread across a distributed filesystem. Instead, the group administrator should be able to manage a central membership list, allowing others to refer to the list as needed. Chirp provides this mechanism in the form of *distributed groups*.

For example, Figure 4 shows a distributed group used to called CSE to represent members of the computer science and engineering department. The group is defined on the host *omega*, and contains two explicit users identified by their Kerberos credentials, as well as any user with a Globus credential beginning with `/O=CSE/` and any hostname ending in `.cse.nd.edu`. Now, suppose that some data stored on another host *alpha* should be accessible only to the CSE group. The owner of the data sets the ACL to contain the following entry:

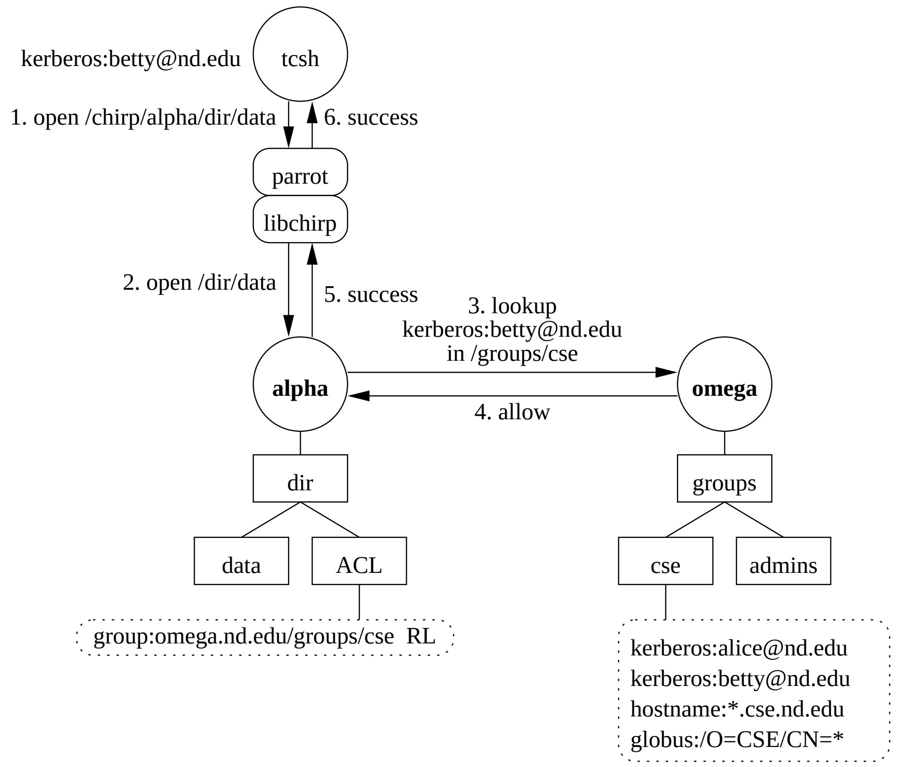


Figure 4. Example of a Distributed ACL Definition

`group:omega.nd.edu/groups/cse`

Now, access control checks on this ACL, if not satisfied by the local entries in the list, will result in a `lookup` RPC to `omega` in order to determine whether the client is a member of the group. By this mechanism, users can establish new groups that cross administrative boundaries, or even accomodate users with different kinds of credentials.

However, distributed groups introduce new problems in performance and semantics. Clearly, adding an RPC to each request for data will increase the latency of all operations. To avoid this, recent `lookup` requests are cached for a time controlled by the server that contains the group, typically five minutes. This allows the group owner to choose the appropriate tradeoff between performance and revocation time.

Caching lookups improves the performance of a single client that repeatedly requests a group, but what about a server that must provide the same data to many members of a group? To avoid multiple distinct `lookups`, the client may also request the entire contents of the group file by issuing an ordinary `getfile` request, which is then cached locally,

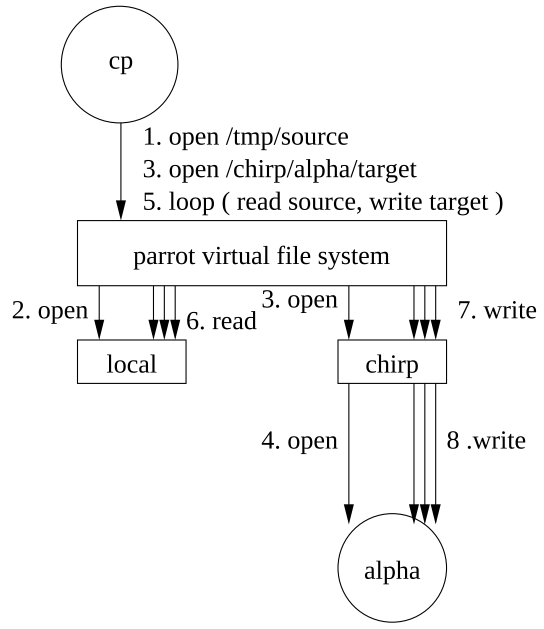

cp
1. open /tmp/source
3. open /chirp/alpha/target
5. loop (read source, write target)
parrot virtual file system
2. open
6. read
3. open
7. write
local
chirp
4. open
8 .write
alpha

Figure 5. An Inefficient Ordinary Copy Command

again for a time controlled by the group owner. Now multiple tests for membership of different subjects may be performed locally.

In some cases, revealing the entire contents of a group may be inappropriate. Either the group list may simply be too big to distribute to all clients (e.g. the list of students on a campus) or the group list itself may be sensitive (e.g. the list of staff with root access.) Again, the owner of the list is in control, and may specify which mechanism is to be used. If the group list is not to be revealed, then `getfile` requests simply fail, and the calling server falls back to individual `lookup` requests.

8. Custom System Calls

Several aspects of the standard Unix interface are not well suited for a grid computing environment. To escape these limitations of Unix, we introduce several new system calls into Parrot that improve both performance and usability. Each of these system calls is exercised by a user-level tool that invokes the capability in a familiar way.

A simple example is the representation of user identity. In Unix, the system call `getuid()` returns the integer identity of the calling user. The integer must be passed to a name lookup service in order to yield a human readable identity. Access control lists in the filesystem associate

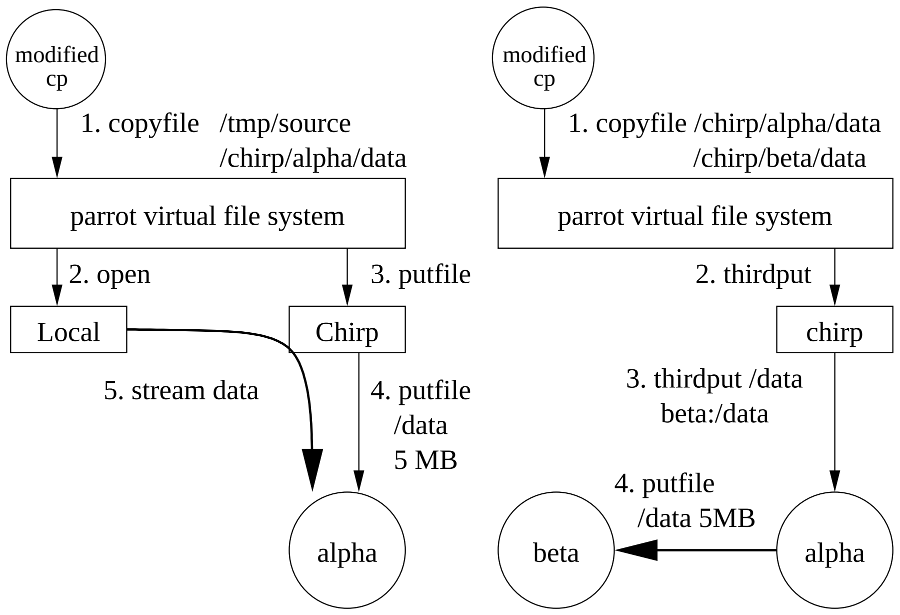

modified cp
1. copyfile /tmp/source /chirp/alpha/data
parrot virtual file system
2. open
3. putfile
Local
Chirp
5. stream data
4. putfile /data 5 MB
alpha
modified cp
1. copyfile /chirp/alpha/data /chirp/beta/data
parrot virtual file system
2. thirdput
chirp
3. thirdput /data beta:/data
4. putfile /data 5MB
beta
alpha

Figure 6. Efficient File Copies with Custom System Calls

rights with integer identities and require the use of the lookup service for implementation.

This model does not map well into any grid identity service. Maintaining a common and consistent user database is difficult enough within one administrative domain, and nearly impossible across domains. Even assuming a common database, a given human may have multiple credentials or authentication methods that change with the resources in use. A user might be known as `kerberos:betty@nd.edu` while using `/chirp/alpha.nd.edu`, but known as `globus:CN=Betty` while using `/chirp/data.grid.org` Likewise, access control lists may refer to wildcards or groups that have no integer representation.

To address this, we introduce several new system calls that manipulate user identities directly as strings, rather than integer identities. For example, the Parrot `whoami` system call (and corresponding tool) queries for the caller's identity *with respect to a given path* in the filesystem. In the case of Chirp, this results in a `chirp_whoami` RPC to obtain a specific server's interpretation of the user's identity. Likewise, the `getacl` and `setacl` system calls manipulate access controls using strings rather than integers.

A more complex example involves copying files. Figure 5 shows how the ordinary `cp` command interacts with the operating system kernel (or Parrot acting as the kernel.) The `cp` tool opens the local file, then the remote file, issues a series of small reads and writes to copy the

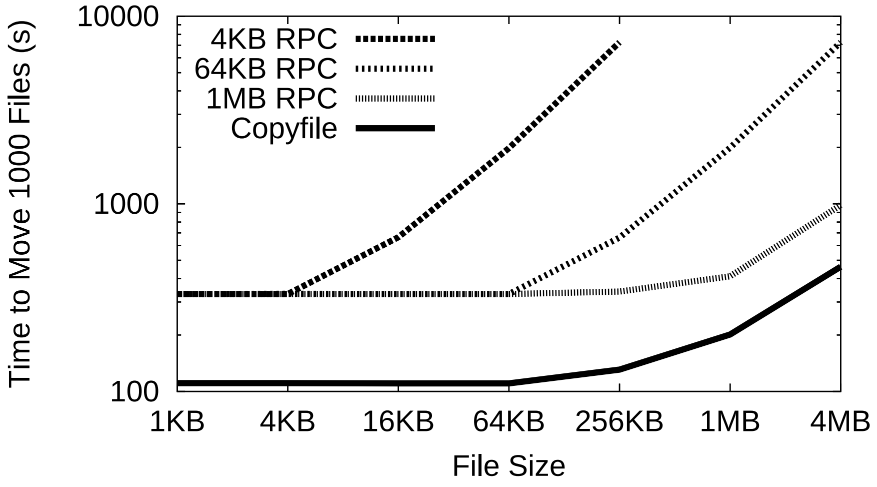


Figure 7. Wide Area File Copy Performance

data, and then closes both files. This is a rather inefficient process for copying small files on high latency networks because a minimum of three network round trips is required. It can also be inefficient for large files, because each small write requires a network round trip.

To improve the performance of file copies, we add a new system call (`copyfile`) to Parrot. This system call accepts two path names and copies all data from one file to another before returning. If a file is copied between local disk and a Chirp server, then Parrot invokes the `putfile` or `getfile` operations to stream the entire file data in one round trip. If a file is copied between two Chirp servers, then Parrot invokes the `thirdput` call to perform a third party transfer between two Chirp servers. If the given path name is actually a directory, the servers will arrange to transfer the entire directory structure without any further instruction from the client. The user is provided with a modified `cp` program that invokes `copyfile`, but falls back to the ordinary behavior if executed outside of Parrot.

Figure 7 shows the significant effect that `copyfile` can have on a high latency wide area network. In this experiment, we emulate a high bandwidth wide-area network by placing a client and server on the same gigabit ethernet switch, and adding an artificial latency of 100ms to the server. We then compare the performance of copying 1000 files of varying sizes from the client to the server, using the ordinary `cp` that results in small RPCs against the modified `cp` that employs `copyfile`. As can be seen, `copyfile` achieves better performance on large files by avoiding multiple round trips, but it also performs better on small files by avoiding `open` and `close`.

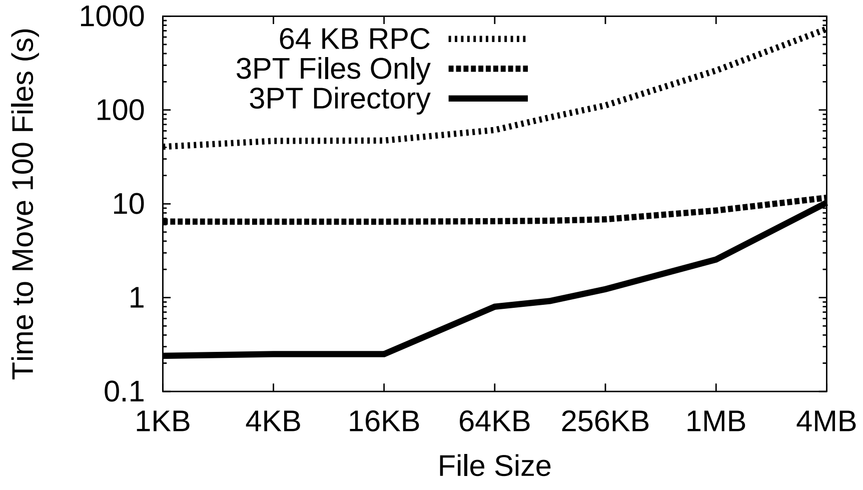


Figure 8. Third Party Transfer Performance

Figure 8 shows the benefits of third-party transfer, again transparently invoked by `cp`. In this experiment, a directory of 100 files of varying sizes is copied between two servers on the same Ethernet switch, directed by a third party over a 30ms wide area network. As expected, RPC is slowest, because all data flows over the wide area network. Performance improves if the third party directs each individual file transfer, but is best when the third party simply indicates the directory name, and allows the entire transfer to be governed by the source.

9. Consistency Semantics

The consistency semantics of a filesystem define when changes made at one node in the system become visible to other nodes in the system. We describe a set of consistency semantics as *strict* if changes are immediately visible to others, while we use *relaxed* to indicate that changes make take some time to become visible. Generally speaking, relaxed consistency semantics provide better performance and availability in the face of failures, but may potentially provide unexpected results for applications written to expect strict semantics.

Most distributed filesystems define a fixed set of consistency semantics that must be employed by all users of the system. For example, NFS [37] implementations typically have many options that can adjust the buffering and caching policies, but they are chosen by the system administrator when a remote filesystem is mounted. AFS [29] provides file snapshot semantics for all participants, requiring that a connection be maintained to receive callbacks.

On the grid, different applications running on the same system may wish to strike the balance between consistency, availability, and performance differently. Each job submitted in a batch of one thousand independent jobs may fairly assume that its inputs will not change during the run and that its outputs need not be stable or visible until the job completes. On the other hand, a workload that is steered by an interactive user may require instant access to new inputs provided at the submitting workstation, and will need to deliver results immediately to the end user. Accordingly, Chirp has three modes for consistency semantics that may be chosen by the end user using command-line arguments:

- **Strict Semantics.** In this mode, no buffering or caching at all is enabled: every I/O operation on a Chirp file server results in a remote procedure call to the server in question. If a server is unreachable, the connection is retried until successful or the failure timeout is reached. This mode yields the same semantics as processes running on a local system, and is suitable for running applications that select small amounts of data from a large collection.
- **File Snapshot Semantics.** In this mode, each file opened by the client is loaded from the remote server and then stored in a disk cache local to the requesting user. While the file is open, no further requests to the server are necessary. If modified, the entire file will be written back to the server on close. If a cached file is re-opened, a remote `stat` operation is performed on the server to test whether the file's size or modification time has changed. (This up-to-date check may also be cached.) A new copy of the file is fetched if needed. This mode yields the same semantics as AFS [29], yielding a snapshot of each file at the time it is opened.
- **Job Session Semantics.** In this mode, files are cached on local disk as in the file snapshot mode, but an up-to-date check is only performed once. Once a file is cached and up-to-date, it is never checked again within the lifetime of the job. This yields semantics suitable for a batch workload, which typically does not expect input files to change as it runs.

In the same spirit, a user may also control the failure timeouts used by Chirp. In a wide area distributed storage system, network outages and other failures are very common. An interactive user exploring Chirp servers will not want to sit idle for five minutes while a server is confirmed to be dead. But, a batch job that will run for hours or days should not fail unnecessarily due to a brief network hiccup. By default, a Chirp session invoked from an interactive terminal will employ a failure timeout of one minute, while a session invoked in a batch context has a timeout of one hour. The user may override this value as needed.

Other systems have made extensive use of callbacks [29, 39, 25] to manage consistency semantics. In this mode, the server is responsible for contacting the client when a file or directory of interest has changed and can no longer be cached. We have chosen not to make use of callbacks for lack of operating system support. Because Chirp exports existing filesystems that may be changed directly by other underlying tools, callbacks would require the operating system to block modifications to the filesystem while it notifies the Chirp server, which collects the callbacks, and then releases the operating system. Unfortunately, each Unix-like operating system implements file change notification in a different way, and none cause the modifier to block until the notification is complete. In addition, the asynchronous nature of callbacks dramatically complicates the networking protocol and software engineering of each component. For these reasons, we have chosen the simpler solution of consistency checks.

Figure 9 shows the performance effects of choosing different consistency semantics on the same Andrew-like benchmark used above in Figure 2. In this case, we ran the benchmark using Parrot while recording all of the remote calls made in each of the three consistency semantics modes. The results show the number of remote I/O calls of each type made to the remote server.

As can be seen, this benchmark is a metadata intensive workload that requires a large number of remote calls to the server. However, as the consistency semantics change from strict unix to file snapshot to job session, the total number of remote calls decreases by a factor of four. From strict to file snapshot, this is done by combining multiple reads and writes on files into local access on cached copies, but a large number of `stat` operations are needed to keep the cached copies consistent. When the semantics are changed to job session, these up-to-date checks are eliminated, but some are replaced by additional `open` calls needed to satisfy `make` searching for a large number of non-existent files.

10. Clustering

So far, we have discussed individual Chirp servers as single file servers for grid computing. However, multiple Chirp servers can also be joined together to provide increased performance or capacity for applications.

remote calls	**strict semantics**	**file snapshot semantics**	**job session semantics**
chmod	50	50	50
chown	9	9	9
close	2,190	1,569	1,569
fstat	2,077	0	0
getdir	198	198	198
lstat	1,833	1,833	1,833
mkdir	40	40	40
open	42,334	1,943	26,285
pread	51,113	1,249	1,249
pwrite	46,630	1,198	1,198
rename	9	9	9
rmdir	40	40	40
stat	3,752	46,084	3,750
unlink	589	589	589
utime	414	414	414
total	**151,278**	**55,216**	**37,764**

Figure 9. Total Remote Calls for Varying Consistency Semantics

We have experience running and managing over 250 Chirp servers totaling 40TB of storage harnessed from workstations and servers at the University of Notre Dame since early 2005.

A cluster of Chirp servers can be joined together in one of two ways. A shared namespace can be used to join multiple servers into a unified filesystem cluster that has a single name and provides location-transparent access to multiple disks. Alternatively, multiple independent Chirp servers can be managed as a loosely coupled cluster that allows the end user to select nodes for location and performance as needed. We consider both kinds of clustering in turn.

10.1. Unified Filesystem Clusters

Figure 10 shows how multiple Chirp servers can be joined into a single large filesystem. One server is employed as a *directory server* that contains only the directory hierarchy of the filesystem. Each file on the directory server records the location of where the actual file is stored on another Chirp server acting as a *data server*. As shown in the

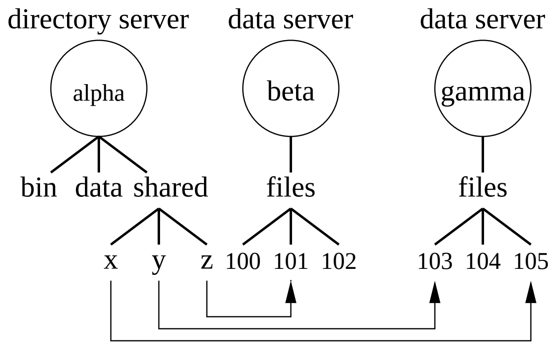


Figure 10. Cluster Filesystem Architecture

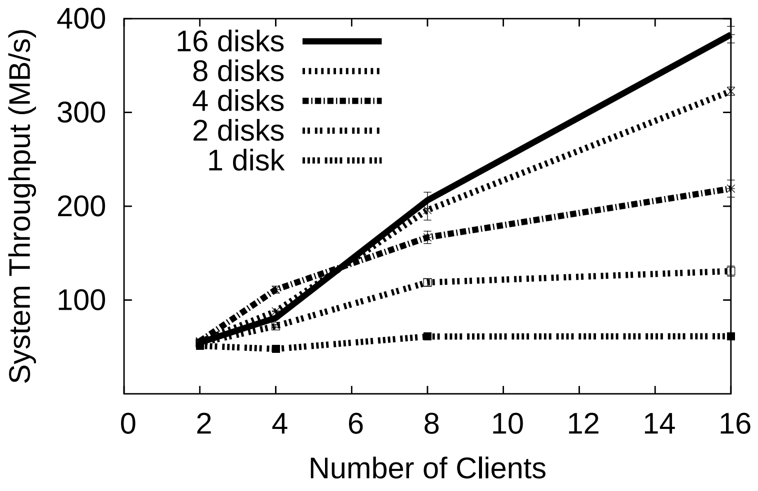


Figure 11. Clustered Filesystem Throughput

shared namespace of Figure 3, distribution is handled transparently via *libchirp*, which contacts all the servers as needed. Such a filesystem can be expanded to arbitrary size, simply by adding additional file servers.

A clustered filesystem can improve the aggregate throughput of many clients accessing the same filesystem simultaneously, by virtue of spreading the read and write load across multiple disks and networks. However, it does not provide any significant performance benefit to individual clients. It is also important to note that the system does not have any internal redundancy, so we recommend that the filesystem be used in conjunction with a conventional tape backup.

Figure 11 shows the throughput available to multiple clients accesing the system simultanously. This experiment employs a 32-node computing cluster with 1-16 nodes used as storage servers, and 1-16 nodes (no overlap) used as clients of the system. For a chosen number

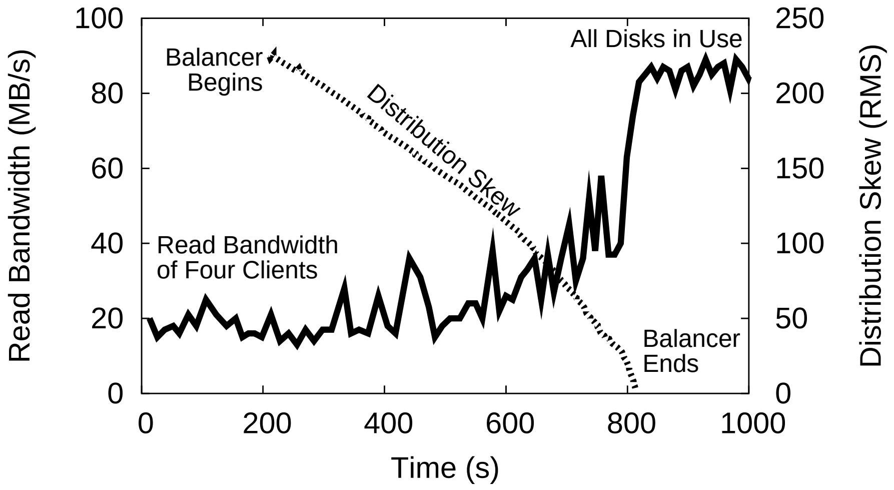


Figure 12. Reconfiguring a Clustered Filesystem

of disks, we populate the filesystem with a large number of 1MB files, and then observe the available aggregate read throughput of multiple clients selecting files at random and reading them sequentially. Adding disks increases the throughput available to multiple clients.

Unlike other storage systems such as RAID [33], a Chirp cluster can easily be reconfigured at runtime. Extra storage servers can be added at run-time by simply adding their names to a configuration file; clients will start writing new files there immediately. Data can be migrated or balanced with a user-level migration tool that examines the filesystem, copies files, and updates the directory server (with appropriate locking) as the filesystem runs.

Figure 12 shows an example of run-time migration. (This is a different cluster than the previous figure, so the absolute performance values do not correspond.) In this experiment, four clients are reading data randomly off of a cluster with two data servers. Six fresh data servers are added to the system and initially have no data. The migration tool is started at time 200 and progressively rebalances files across the system. The *distribution skew* of the system is shown as the root mean squared deviation from an equal number of files on each server. As can be seen, the action of the migration tool steadly increases the throughput of the system to four times the initial value.

10.2. Loosely Coupled Storage Clusters

Location transparency is not always desirable in a distributed system. For many grid applications, it is important to know where data is lo-

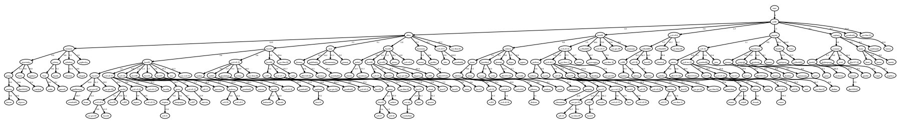


Figure 13. Example Spanning Tree for File Distribution

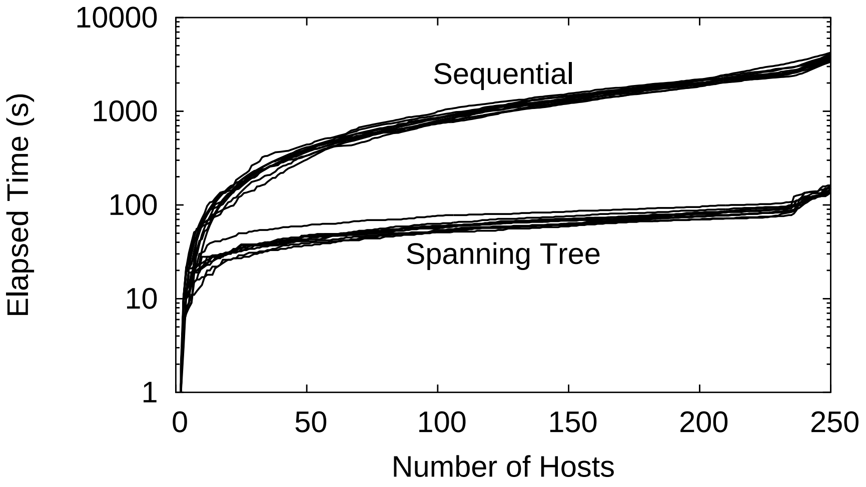
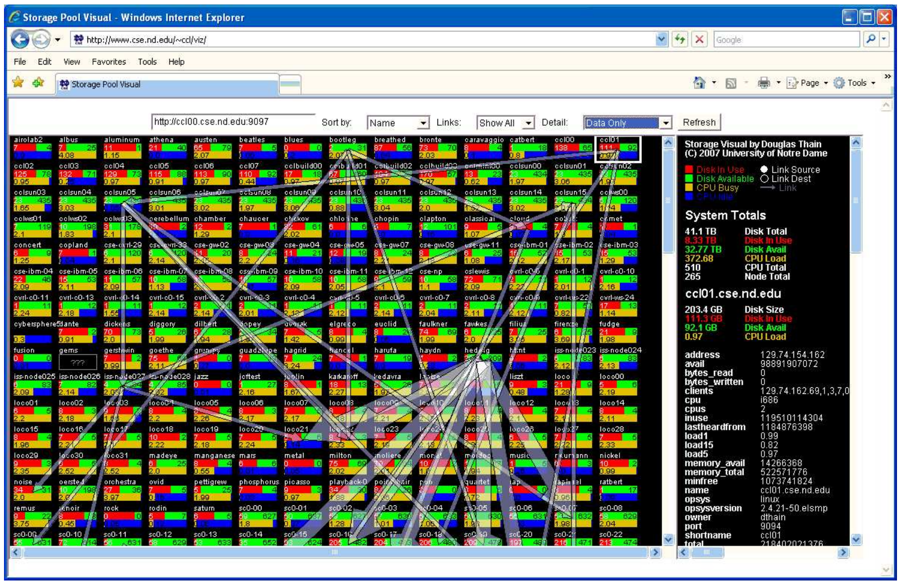


Figure 14. File Distribution Performance

cated so as to achieve good performance. A process that requires a large amount of input data will run fastest when running on a CPU close to the relevant disk. For such applications, a loosely coupled cluster is more appropriate. In this setting, we can replicate data at many individual servers, and then allow jobs to pick the closest copy of the data to use. In our 250-node storage cluster, it is common for users to replicate a data set to *all servers*, and then submit batch jobs that assume data is available everywhere.

However, copying a large file to 250 nodes sequentially can be very time consuming, and may even take longer than the intended computation. Exploiting the third party transfer described above, we have created a tool `chirp_distribute` that constructs a spanning tree to distribute the file in $O(log(n))$ time. `chirp_distribute` operates in a greedy fashion as follows. We assume that a single copy of the data exists on a source host. The source server is directed to transfer its data to a random target. Then, both hosts are used as sources for two new targets in parallel. This process continues until the data arrives at all hosts. Figure 13 shows an example spanning tree generated by this method. Note that it is imbalanced due to varying transfer performances between hosts.


Storage Pool Visual - Windows Internet Explorer
http://www.cse.nd.edu/~ccl/viz/
File Edit View Favorites Tools Help
Storage Pool Visual
Page Tools
Refresh

http://ccl00.cse.nd.edu:9097 Sort by: Name Links: Show All Detail: Data Only

```
Storage Visual by Douglas Thain
(C) 2007 University of Notre Dame

🟥 Disk In Use        ● Link Source
🟩 Disk Available     ○ Link Dest
🟨 CPU Busy           → Link
🟦 CPU Idle

System Totals
41.1 TB      Disk Total
8.33 TB      Disk In Use
32.77 TB     Disk Avail
372.68       CPU Load
510          CPU Total
265          Node Total

ccl01.cse.nd.edu
203.4 GB     Disk Size
111.3 GB     Disk In Use
92.1 GB      Disk Avail
0.97         CPU Load

address          129.74.154.162
avail            98891907072
bytes_read       0
bytes_written    0
clients          129.74.162.69,1,3,7,0
cpu              i686
cpus             2
inuse            119510114304
lastheardfrom    1184876398
load1            0.99
load15           0.82
load5            0.97
memory_avail     14266368
memory_total     522571776
minfree          1073741824
name             ccl01.cse.nd.edu
opsys            linux
opsysversion     2.4.21-50.elsmp
owner            dthain
port             9094
shortname        ccl01
total            218402021376
```

Figure 15. Chirp Cluster Monitor

Figure 14 compares the performance of sequential distribution versus a spanning tree. A 100 MB file is distributed to 250 servers. Both sequential and spanning tree are run 10 times, each generating a curve. As expected, the random spanning tree is an order of magnitude faster. We suspect that even better performance is possible if the spanning tree is chosen more carefully, but we leave this as future work.

Managing a large cluster of storage servers requires some infrastructure. Each active server periodically sends a short message describing its vital statistics (available space, memory, etc.) and active network connections to the global catalog server shown in Figure 1. This information is published in several formats as a web page which serves as an information source for several high-level information tools such as the system monitor shown in Figure 15. Visual network connections and color-coded resources allows the system manager to quickly unexpected identify problems or abusive users.

The cluster storage manager does not necessarily have a privileged login on all machines in the cluster, each of which are owned and managed by different people and departments. However, the manager must still have some capability to fix problems remotely. To provide this, each server is configured to recognize a *storage superuser* that has the capability to list files and modify ACLs in any directory. The storage superuser is typically a GSI subject, so that a reasonable degree of communication security can be guaranteed.

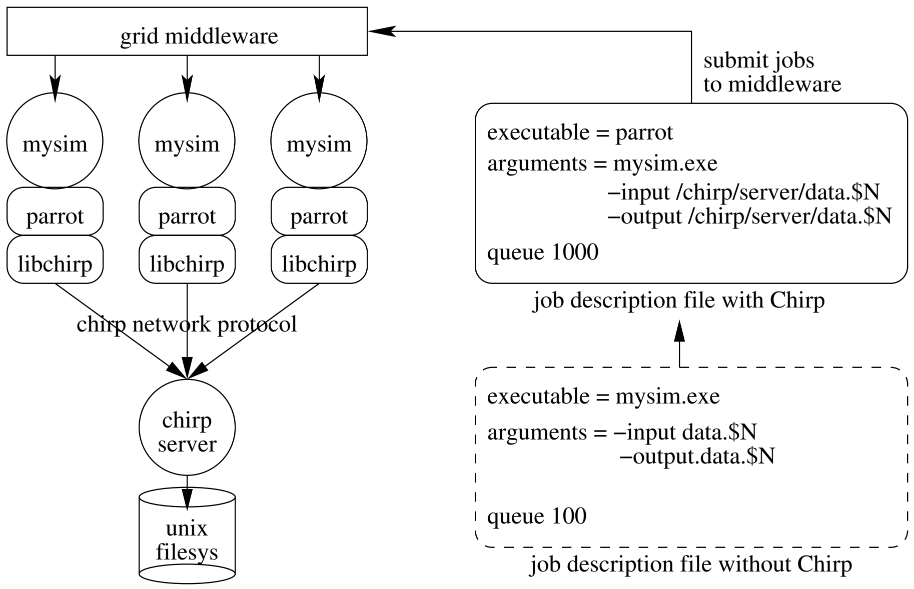


Figure 16. Typical Use of Chirp with Existing Grid Middleware

The storage superuser also has the ability to invoke a *storage audit* on any individual machine. This audit traverses the local filesystem and produces a report listing the storage consumption by individual users. When done in parallel, the data are combined to produce a daily listing of resource consumption by user and location. Because this is done in parallel, the state of the entire 250-disk system can be audited in a matter of minutes.

11. Chirp on the Grid

Figure 10.2 shows how Chirp would typically be used with existing computational grid infrastructures. Suppose that the user is accustomed to submitting jobs to run the executable `mysim.exe` on a local cluster. In order to dispatch them to the grid, the user starts a Chirp server on the machine that contains the necessary data and executables, adjusts the batch submit file to run `parrot` with `mysim.exe` as the argument, and adjusts the pathnames to refer to the new Chirp server. Now, the jobs can be submitted to execute on any node on the grid, and wherever they happen to run, the jobs will connect back to the Chirp server to perform I/O as needed.

Note that this scenario only relies on the ability of the grid computing system to execute Parrot as a job on some approprate CPU. Chirp is decoupled from the other details of the grid computing system, and thus can function with any kind of middleware used to dispatch jobs to CPUs. Because Chirp performs I/O on demand, it also functions independently of grid workflow systems such as Pegasus [19].

12. Applications

We conclude by describing three applications that have employed the Chirp filesystem for data intensive grid computing. Each application employs the filesystem in a different way: one as a filesystem bridge, one as a filesystem cluster, and another as a loosely coupled cluster.

12.1. Personal Filesystem Bridge for Genomic Searches

BLAST [11] is a common data-intensive application used in the life sciences. It is used to compare a large number of small (10-100 byte) genomic sequences against large (several GB) standard databases of genomic data. It is most commonly run on clusters where all nodes share an NFS-mounted central server on a fast network. Typically, the administrator installs a selection of databases describing different organisms, each database consisting of a handful of index and data files. The user runs the `blastall` program, which invokes a sub-program and accesses databases given by command line arguments.

BLAST would seem to be a natural application to run on a computational grid, but it is not easy to do so. In a traditional stage-in, stage-out model, the user must be careful to identify the precise set of sub-programs and data files to be sent to each remote site. But, because the databases are so large, it would be inefficient to the point of useless to transfer data along with every single job to be run, so the user must stage the needed databases to the head node of each cluster. This procedure must be repeated every time the database changes or a new cluster is harnessed. Few users are inclined to manage this procedure manually.

Using Chirp, we can build a secure distributed file system that makes it easy to run BLAST jobs without any site-specific configuration. Figure 17 shows an example of a system we have constructed that runs BLAST jobs across two different grid systems at the University of Notre Dame and Purdue University, all accessing a file server deployed at Notre Dame.

The batch job is simply a script that loads executables and data from the `/blast` directory, which is directed to the file server using a private name space. Then, Parrot and the script are submitted to each batch system along with the user's GSI credentials. The script fetches

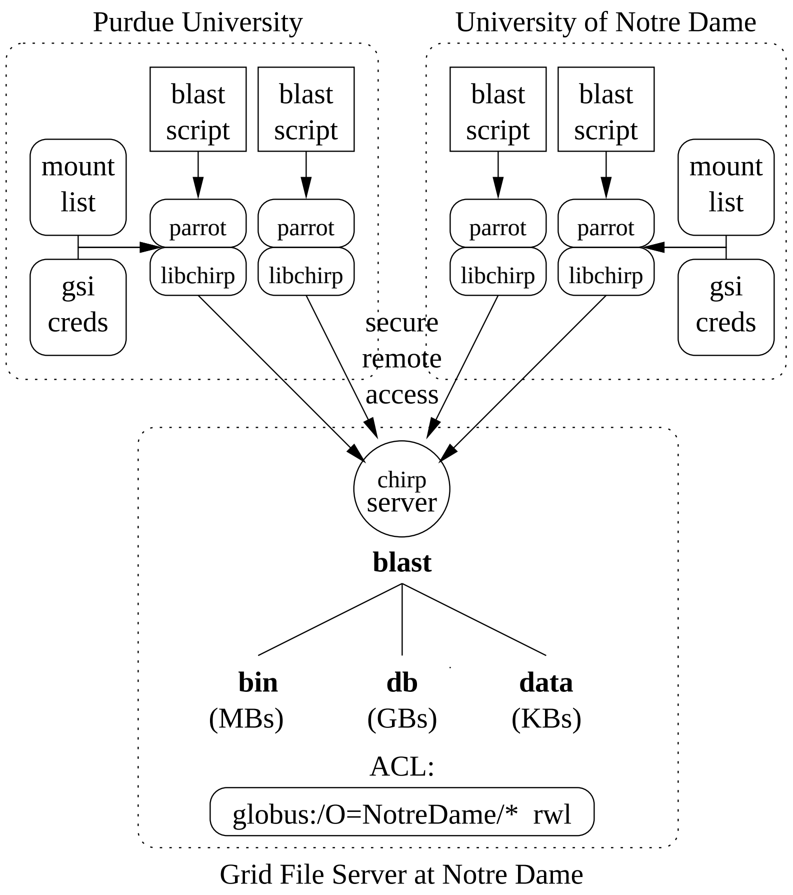


Figure 17. Running BLAST on Multiple Grids with Chirp

both executables and data via Chirp, caches them on the local disk, and runs the search. Further jobs that run on the same node can take advantage of the cached data.

To consider the performance of this system, we first examine a single BLAST job that searches for a match to one sequence within the standard non-redundant human protein database known as `nr` and writes the output to remote storage. We measure the total number of remote I/O calls, the amount of data read and written, and the wall clock time necessary to complete one search using strict semantics (direct remote access) and job session semantics, with cold and warm caches. The results are shown in Figure 18.

As can be seen, the absolute runtime for one job is fastest with strict semantics: it is faster to stream data stored in memory on a remote machine than to read it off a local disk. However, the scalability of these approaches is very different: the strict semantics would quickly

	strict semantics	**session: cold cache**	**session: warm cache**
remote calls	23249	23119	45
data read	1.5 GB	1.5 GB	none
data written	172 KB	172 KB	172 KB
best runtime	38 sec	84 sec	62 sec

Figure 18. Performance of a Single BLAST

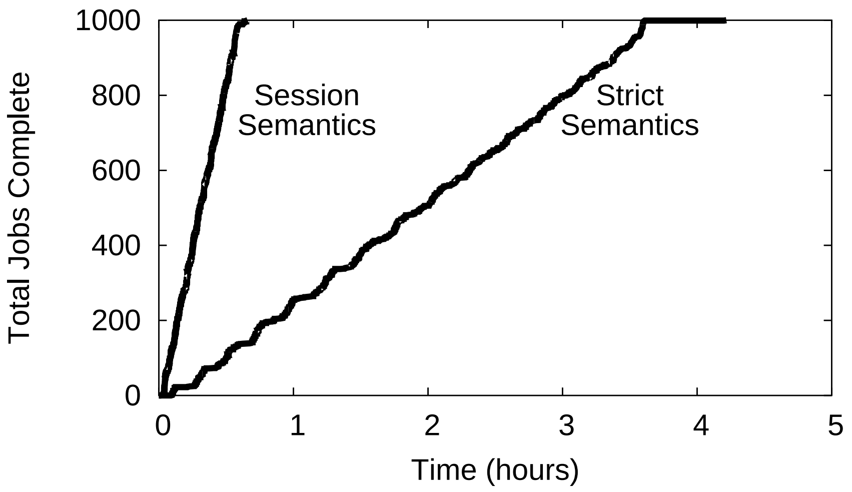


Figure 19. Performance of a BLAST Batch

overwhelm a central file server, while the snapshot semantics can cache large files for as long as they do not change on the server.

To demonstrate this, we submitted a batch of 1000 BLAST jobs, each searching for 10 sequences within the `nr` database to the system described above. In both cases, 70 machines spread across both systems were available for our use. The results are shown in Figure 19. As expected, the session semantics resulted in much higher job throughput, and lower average runtimes. For the strict semantics, one may see a stair-step behavior indicating intervals in which all processes were blocked waiting on the central server for I/O.

By employing Chirp as a distributed filesystem across the grid, the user can take advantage of multiple clusters just as easily as using one cluster. The user is saved from the inconvience of manual data staging to cluster heads, and from the untenable performance overhead of staging data on each job execution.

12.2. Clustered Storage for Physics Data Analysis

The Gamma Ray Astrophysics at Notre Dame (GRAND) [36] project at Notre Dame studies stellar sources of gamma rays by measuring muon showers created as gamma rays interact with the Earth's atmosphere. The project detector consists of an array of 64 detectors arranged in a field on the Notre Dame campus, producing about 50MB of data per hour, 365 days a year.

As a small-scale physics project, GRAND does not have a large dedicated computing center to support its activities. The project records new data on to commodity disks, and then to inexpensive offline tape as the disks fill up. As a result, only the most recently-recorded data is easily available for analysis. Any historical study on archived data requires long hours of sorting through tapes.

However, using Chirp, we can harness existing conventional workstations and servers to create an inexpensive but high capacity clustered filesystem that allows for efficient historical study in ways not possible with a tape archive. To meet these requirements, we created a clustered filesystem of 32 nodes, totalling about 16 TB of storage. As data is acquired, it is still migrated to offline tape for backup, but is also duplicated into the storage cluster. The capacity of the cluster is sufficient for storage, at the current rate, of more than 13 years' worth of GRAND data. The cluster has been actively collecting new data since early 2006, and loading of historical data is in progress.

The filesystem cluster has worse performance than a single locally attached disk, but is far better than tape for large amounts of data. The latency of a write to the local disk through the system buffer cache is 0.04 ms, while latency to the storage cluster is about 0.3ms over the campus-area network. However, as shown above, the aggregate data throughput scales up with the number of servers. The cluster regularly supports several students working simultaneously on historical data analysis not possible under the old system.

12.3. Data Replication for Biometric Image Comparisons

Biometric researchers at the University of Notre Dame are designing and evaluating new algorithms for face recognition. Abstractly speaking, a face recognition algorithm is a function that takes two images as inputs and produces an output score between zero and one, indicating the similarity of the two images. To evaluate the quality of such a function systematically, it must be used to compare all possible pairs of images from a known gallery, producing a *similarity matrix* of results that characterizes the quality of the function.

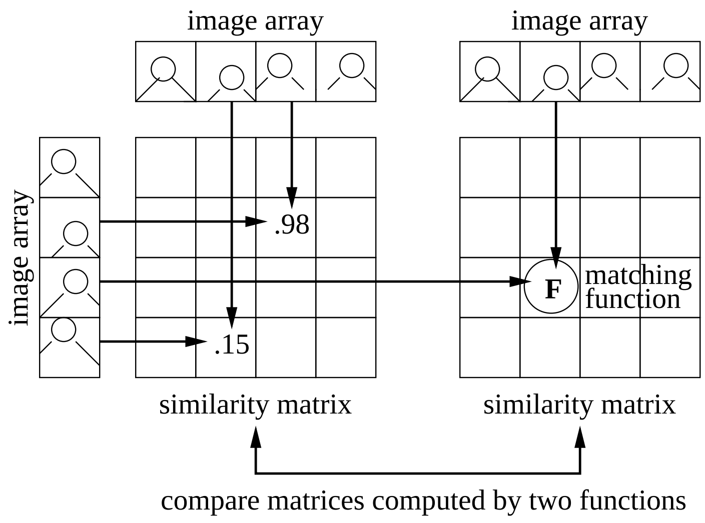


Figure 20. Image Comparison for Biometrics

Figure 20 shows the structure of this computation. A typical workload of this kind consists of 4000 images of about 256 KB each taken from the Face Recognition Grand Challenge (FRGC) [34] data set, all compared to each other with the ICP [17] algorithm. Each match compares 14 regions of the face, each region requiring about 1 second of computation. This requires about 2500 CPU-days of computation in total, so the more CPUs that can be applied to the problem, the faster it can be completed. We wish to apply about 250 CPUs to the problem so that it can be completed in about 10 days.

Unfortunately, each CPU added to the system increases the I/O load. 250 CPUs simultaneously loading 1GB of data will overload both our networks and any central storage server. To avoid this problem, we exploit the distributed I/O capacity of our loosely coupled storage cluster. For each run, we build a spanning tree to distribute the input data to each CPU. Unfortunately, not all CPUs are willing to donate enough local space to store all data sets of interest. So, we must make a smaller number of copies to hosts that have available space, and then use the filesystem interface of Chirp to access the needed data at runtime from each computation node. In this way, data is distributed to all CPUs in two stages.

To get reasonable performance, we must distribute the data appropriately. Figure 21 shows the aggregate throughput of five data servers in several configurations, each serving data to up to eighty clients simultaneously. **A** is a single server distributing the archive as a single 500 MB file, and **B** is a set of five servers doing the same. It is no

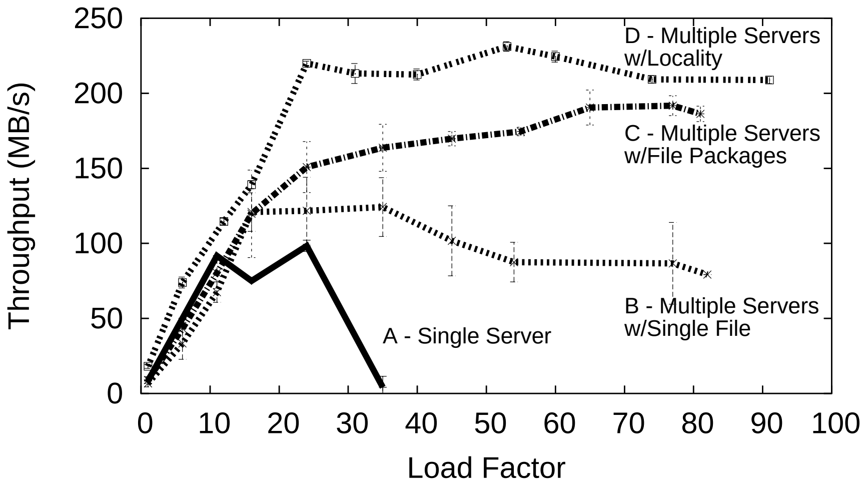


Figure 21. Aggregate Throughput Using Five Replicated Servers

surprise that more servers can handle more load. However, when many clients access the same server at once, responsiveness becomes so poor that clients make no progress, and eventually timeout and fail. In **C** the files are grouped together into packages of 50MB, which are loaded by clients as needed. In **D** the same scheme is used, but each job chooses a server on the same physical cluster.

The inherent limitation of a single data server is evident: any single data server has limited resources, whether the limiting factor is storage, memory, or network link bandwidth. For this specific workload, smaller pieces of the dataset achieved higher aggregate throughput than the dataset as a whole. Utilizing cluster locality, where available, also gave a significant boost in performance, both peak throughput and scalability, over the random choice.

Using this system, the biometrics researchers were able to execute several workloads in several days instead of several months. They completed four separate data-intensive evaluations, varying from 85 CPU-hours to 1300 CPU-hours and 700 GB to 5.3 TB of total I/O. The highest amortized I/O requirement for any evaluation was 40 Mb/CPU-second, including transfers wasted due to failures. In another run, the storage system was able to serve a start-up burst at over 11.2 Gbps over a one-minute interval using 30 data servers for approximately 250 worker nodes.

13. Conclusion

Although distributed file systems have long been a focus of study, we have argued that the grid is a sufficiently different environment that it requires a new design for distributed filesystems. We have described the Chirp filesystem, which provides unique services for deployment, naming, consistency, security, and clustering that are particular to grid applications. We demonstrate three applications that employ these services in different ways to exploit the full power of computing grids.

The Chirp software is available for download at:

`http://www.cse.nd.edu/~ccl/software`

Acknowledgements

We thank the anonymous reviewers for their help in improving this paper. We thank James Fitzgerald and Paul Madrid, who contributed to the implementation of Chirp. We thank Preston Smith for deploying Chirp, and assisting us with the Purdue Condor pool. We thank the many users and testers of Chirp, particularly Justin Wozniak, Paul Brenner, Aaron Striegel, Jesus Izaguirre, Tim Faltemier, Patrick Flynn, Chris D'Andrea, John Poirier, Christophe Blanchet, Remi Mollon, Gilbert Deleage, Mark Calleja, and Sander Klous. This work was supported in part by National Science Foundation grants CCF-06-21434 and CNS-06-43229.

References

1. 'Enabling Grids for E-SciencE'. http://www.eu-egee.org.
2. 'Filesystem in User Space'. http://sourceforge.net/projects/fuse.
3. 'Global Grid Forum'. http://www.ggf.org.
4. 'L-Store: Logistical Storage'. http://www.lstore.org.
5. 'SlashGrid'. http://www.gridsite.org/slashgrid.
6. 'TeraGrid'. http://www.teragrid.org.
7. 'The Open Science Grid'. http://www.opensciencegrid.org.
8. Alexandrov, A., M. Ibel, K. Schauser, and C. Scheiman: 1998, 'UFO: A personal global file system based on user-level extensions to the operating system'. *ACM Transactions on Computer Systems* pp. 207–233.
9. Allcock, W., J. Bresnahan, R. Kettimuthu, and J. Link: 2005, 'The Globus eXtensible Input/Output System (XIO): A Protocol Independent IO System for the Grid'. In: *Workshop on Middleware for Grid Computing*. Melbourne, Australia.

10. Allcock, W., A. Chervenak, I. Foster, C. Kesselman, and S. Tuecke: 2000, 'Protocols and Services for Distributed Data-Intensive Science'. In: *Proceedings of Advanced Computing and Analysis Techniques in Physics Research*. pp. 161–163.
11. Altschul, S., W. Gish, W. Miller, E. Myers, and D. Lipman: 1990, 'Basic local alignment search tool'. *Journal of Molecular Biology* **3**(215), 403–410.
12. Andrews, P., P. Kovatch, and C. Jordan: 2005, 'Massive High-Performance Global File Systems for Grid Computing'. In: *Supercomputing*. Seattle, WA.
13. Baru, C., R. Moore, A. Rajasekar, and M. Wan: 1998, 'The SDSC Storage Resource Broker'. In: *Proceedings of CASCON*. Toronto, Canada.
14. Batsakis, A. and R. Burns: 2004, 'Cluster Delegation: High-Performance Fault-Tolerant Data Sharing in NFS'. In: *High Performance Distributed Computing*.
15. Beck, M., T. Moore, and J. Plank: 2002, 'An End-to-End Approach to Globally Scalable Network Storage'. In: *ACM SIGCOMM*. Pittsburgh, Pennsylvania.
16. Bent, J., V. Venkataramani, N. LeRoy, A. Roy, J. Stanley, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and M. Livny: 2002, 'Flexibility, Manageability, and Performance in a Grid Storage Appliance'. In: *IEEE Symposium on High Performance Distributed Computing*. Edinburgh, Scotland.
17. Besl, P. and N. McKay: 1992, 'A method for registration of 3-D shapes'. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **14**.
18. Bester, J., I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke: 1999, 'GASS: A Data Movement and Access Service for Wide Area Computing Systems'. In: *6th Workshop on I/O in Parallel and Distributed Systems*.
19. Deelman, E., G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, B. Berriman, J. Good, A. Laity, J. Jacob, and D. Katz: 2005, 'Pegasus: A Framework for Mapping Complex Scientific Workflows onto Distributed Systems'. *Scientific Programming Journal* **13**(3).
20. Figueiredo, R., N. Kapadia, and J. Fortes: 2001, 'The PUNCH Virtual File System: Seamless Access to Decentralized Storage Services in a Computational Grid'. In: *IEEE High Performance Distributed Computing*. San Francisco, CA.
21. Foster, I., C. Kesselman, J. Nick, and S. Tuecke: 2002, 'Grid Services for Distributed System Integration'. *IEEE Computer* **35**(6).
22. Foster, I., C. Kesselman, G. Tsudik, and S. Tuecke: 1998, 'A Security Architecture for Computational Grids'. In: *ACM Conference on Computer and Communications Security*. San Francisco, CA, pp. 83–92.
23. Foster, I., C. Kesselman, and S. Tuecke: 2001, 'The Anatomy of the Grid: Enabling Scalable Virtual Organizations'. *Lecture Notes in Computer Science* **2150**.
24. Ganguly, A., A. Agrawal, P. O. Boykin, and R. J. Figueiredo: 2007, 'WOW: Self Organizing Wide Area Overlay Networks of Workstations'. *Journal of Grid Computing* **5**(2).
25. Gray, C. and D. Cheriton: 1989, 'Lease: An efficient fault-tolerant mechanism for distributed file cache consistency'. In: *Twelfth ACM Symposium on Operating Systems Principles*. pp. 202–210.
26. Grimshaw, A., W. Wulf, et al.: 1997, 'The Legion Vision of a Worldwide Virtual Computer'. *Communications of the ACM* **40**(1), 39–45.
27. Hemmes, J. and D. Thain: 2006, 'Cacheable Decentralized Groups for Grid Resource Access Control'. In: *IEEE Conference on Grid Computing*. Barcelona, Spain.
28. Honeyman, P., W. A. Adamson, and S. McKee: 2005, 'GridNFS: Global Storage for Global Collaboration'. In: *Local to Global Data Interoperability*.

29. Howard, J., M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West: 1988, 'Scale and Performance in a Distributed File System'. *ACM Trans. on Comp. Sys.* **6**(1), 51–81.
30. Jones, M.: 1993, 'Interposition Agents: Transparently Interposing user Code at the System Interface'. In: *14th ACM Symposium on Operating Systems Principles*. pp. 80–93.
31. Li, W., J. Liang, and Z. Xu: 2003, 'VegaFS: A prototype for file sharing crossing multiple domains'. In: *IEEE Conference on Cluster Computing*.
32. Moretti, C., T. Faltemier, D. Thain, and P. Flynn: 2007, 'Challenges in Executing Data Intensive Biometric Workloads on a Desktop Grid'. In: *Workshop on Large Scale and Volatile Desktop Grids*. Long Beach, CA.
33. Patterson, D. A., G. Gibson, and R. Katz: 1988, 'A Case for Redundant Arrays of Inexpensive Disks (RAID)'. In: *ACM SIGMOD international conference on management of data*. pp. 109–116.
34. Phillips, P. and et al.: 2005, 'Overview of the face recognition grand challenge'. In: *IEEE Computer Vision and Pattern Recognition*.
35. Plank, J., M. Beck, W. Elwasif, T. Moore, M. Swany, and R. Wolski: 1999, 'The Internet Backplane Protocol: Storage in the network'. In: *Network Storage Symposium*.
36. Poirier, J., G. Canough, J. Gress, S. Mikocki, and T. Rettig: 1990. *Nuclear Physics B Proceedings Supplements* **14**, 143–147.
37. Sandberg, R., D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon: 1985, 'Design and implementation of the Sun network filesystem'. In: *USENIX Summer Technical Conference*. pp. 119–130.
38. Shoshani, A., A. Sim, and J. Gu: 2002, 'Storage Resource Managers: Middleware Components for Grid Storage'. In: *Nineteenth IEEE Symposium on Mass Storage Systems*.
39. Srinivasan, V. and J. Mogul: 1989, 'Spritely NFS: Experiments with Cache Consistency Protocols'. In: *ACM Symposium on Operating Systems Principles*.
40. Steiner, J., C. Neuman, and J. I. Schiller: 1988, 'Kerberos: An authentication service for open network systems'. In: *Proceedings of the USENIX Winter Technical Conference*. pp. 191–200.
41. Stone, N. and et al.: 2006, 'PDIO: High Performance Remote File I/O for Portals Enabled Compute Nodes'. In: *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*. Las Vegas, NV.
42. Tatebe, O., N. Soda, Y. Morita, S. Matsuoka, and S. Sekiguchi: 2004, 'Gfarm v2: A grid file system that supports high-performance distributed and parallel data computing'. In: *Computing in High Energy Physics (CHEP)*.
43. Thain, D.: 2006, 'Operating System Support for Space Allocation in Grid Storage Systems'. In: *IEEE Conference on Grid Computing*.
44. Thain, D., S. Klous, J. Wozniak, P. Brenner, A. Striegel, and J. Izaguirre: 2005, 'Separating Abstractions from Resources in a Tactical Storage System'. In: *IEEE/ACM Supercomputing*.
45. Thain, D. and M. Livny: 2000, 'Bypass: A tool for building split execution systems'. In: *IEEE High Performance Distributed Computing*. Pittsburg, PA.
46. Thain, D. and M. Livny: 2003, 'Parrot: Transparent User-Level Middleware for Data-Intensive Computing'. In: *Proceedings of the Workshop on Adaptive Grid Middleware*. New Orleans.

47. Thain, D. and C. Moretti: 2007, 'Efficient Access to Many Small Files in a Filesystem for Grid Computing'. In: *IEEE Conference on Grid Computing*. Austin, TX.
48. Thain, D., T. Tannenbaum, and M. Livny: 2003, 'Condor and the Grid'. In: F. Berman, G. Fox, and T. Hey (eds.): *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley.
49. Vazhkudai, S., X. Ma, V. Freeh, J. Strickland, N. Tammineedi, and S. Scott: 2005, 'FreeLoader: Scavenging Desktop Storage Resources for Scientific Data'. In: *Supercomputing*. Seattle, Washington.
50. Walker, E.: 2006, 'A Distributed File System for a Wide-Area High Performance Computing Infrastructure'. In: *USENIX Workshop on Real Large Distributed Systems*. Seattle, WA.
51. Weil, S. A., S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn: 2006, 'Ceph: A Scalable, High-Performance Distributed File System'. In: *USENIX Operating Systems Design and Implementation*.