Accessing Files in an Internet: The Jade File System

Herman C. Rao and Larry L. Peterson*

Abstract

This paper introduces the Jade File System, which provides a uniform way to name and access files in an internet environment. Jade is a logical system that integrates a heterogeneous collection of existing file systems, where by heterogeneous we mean that the underlying file systems support different file access protocols. Because of autonomy, Jade is designed under the restriction that the underlying file systems may not be modified. In order to avoid the complexity of maintaining an internet-wide, global name space, Jade permits each user to define a private name space. Jade's name space supports two novel features: It allows multiple file systems to be mounted under one directory, and it permits one logical name space to mount other logical name spaces. A prototype of Jade has been implemented to examine and validate its design. The prototype consists of interfaces to the Unix File System, the Sun Network File System, and the File Transfer Protocol. This paper gives an overview of Jade's design, and reports our experiences designing and implementating a large scale file system.

1 Introduction

Internets like the National Research and Education Network (NREN) provide an opportunity to unite geographically dispersed users and computing resources into an integrated computing environment. The key to providing such an environment is to develop mechanisms that allow users to transparently name and access the available resources. This paper presents a new distributed file system, called Jade, that addresses the problem of naming and accessing internet-wide resource in the context of file systems.

Many distributed file system designs have been proposed and implemented over the last decade; many of these systems are surveyed elsewhere [Levy90, Saty89, Svob84]. These systems have addressed several important issues, including naming and transparency, consistency and availability, remote access methods, and fault tolerance. Designing an internet-wide file system, however, introduces a new issue: *scalability*. We

*This work supported in part by National Science Foundation Grant CCR-8811423 and NCR-9005028, and National Aeronautics and Space Administration Grant NCC-2-561. Rao's current address: AT&T Bell Laboratories, Murray Hill, NJ 07974. Peterson's current address: Department of Computer Science, University of Arizona, Tucson, AZ 85721.

envision an internet file system encompassing millions of participants, where by participants we are referring to servers maintaining resources, and clients consuming resources. In such an environment, scalability has four dimensions:

- Size—number of participants. It is estimated that the NREN currently includes more than 4,000 active networks and 750,000 computers, and it has been growing exponentially over the past ten years. Clearly, an internet file system based on static assumptions that the number of participants is bounded by a constant will not scale well.
- Area—distance between the participants. In the current NREN, it is not uncommon to see 100 msec round trip times to a nearby site and 400 msec round trip times to a site across the country, making the latency between any two participants in an an internet file system two orders of magnitude greater than in a file system designed for a local area network.
- Heterogeneity—diversity of software and hardware of participants. In the file system arena alone, we estimate that there are at least 20 different distributed file systems in use throughout the NREN. As a consequence, an internet file system must accommodate a wide collection of systems.
- Autonomy—independence and self-determination of individual participants. The major goal of the NREN is to support connectivity among a collection of autonomous organizations. It is important that an internet file system make minimal intrusions on this autonomy.

Jade addresses these four dimensions by providing a logical system that integrates a collection of existing file systems. These underlying file systems may be heterogeneous in the sense that they support different file access protocols for communications between file servers and their users. In addition, Jade supports the following features:

- Jade is designed under the restriction that the underlying file systems may not be modified in software nor changed in administration. The underlying file systems treat an instance of Jade as a regular file system user without any special privilege.
- Rather than providing a global name space, Jade permits each user to define a private name space. A given user has the same view of the file system, regardless of what machine he or she is using.
- To facilitate file sharing, Jade allows one logical file system to be mounted into another Jade file system, in the same way that a physical file system can be mounted into Jade. This allows each user to transparently name and access files *through* another user's name space.
- To support a variety of access paradigms and to encourage collaboration among users, Jade refines the mount operation provided by Unix-like file systems to allow multiple file systems, either logical or physical, to be mounted under a single directory. This feature is called the *multiple mount*.
- Because of the high cost of accessing remote servers, Jade employs whole file caching. Opening a file causes it to be cached in its entirety locally. Reads and writes are directed to the cached copy. Because of the autonomy restriction, however, Jade inherits the file consistency problems of the underlying file systems.

This paper makes two contributions. First, it describes Jade's overall design, highlighting its novel features. This discussion is presented in Section 2. Second, it relates our experiences implementing and using a file system on an internet level. Towards this end, Section 3 reports on the performance of Jade and Section 4 discusses various issues related to the four dimensions of scalability outlined above.

2 Design Overview

This section sketches the design of the Jade file system. It abstractly defines the underlying physical file systems upon which the Jade file system is built and introduces Jade's salient features.

2.1 Physical File Systems

Jade adopts a simple model of the underlying physical file systems. A physical file system provides two services: It maps file names into file handles, and it stores and retrieves file data associated with a given file handle. Each physical file system is identified by the network address of the host where the file system resides and a host-specific identifier for the file system. We assume that each physical file system represents files as non-typed byte-streams.

An access protocol is the key to the services provided by a physical file system. Examples of access protocols include the protocol used by Sun's Network File System (NFS) [Sand85], the protocol defined by the Andrew File System (AFS) [Howa88], and the File Transfer Protocol (FTP) [Post85]. The system interface, supported by the Unix operating system to access files on the local disk, is considered as a protocol for the Unix File System (UFS). Unless otherwise stated, we use the terms "NFS", "AFS", and "UFS" to refer to the access protocols for the Network File System, the Andrew File System, and the Unix File System respectively, not the file system itself.

The access protocol not only provides the key to accessing the physical file system, but it also hides the heterogeneity of the system on which the physical file system is located. That is, once the proper access protocol is available, it is possible to access the services provided by a physical file system without regard to the machine type or the operating system. Thus, Jade has only to deal with a heterogeneous collection of access protocols.

Jade defines a uniform interface to accommodate this collection of access protocols. An access protocol-specific agent maps operations defined by the interface into functions provided by the access protocol. Table 1 lists operations defined by the interface.

Fetch	retrieve an entire file from a physical file system.
Restore	store data back to a file in a physical file system.
GetEntries	get entries in a directory in a physical file system.
RemoveEntry	remove an entry in a directory in a physical file system.
GetAttr	return the attributes associated with a file or directory in a physical file system.
SetAttr	set attributes of a file or directory in a physical file system.
Connect	connect the server that supports a physical file system.
Disconnect	disconnect the server.
MakeDir	create a new directory in a physical file system.
RemoveDir	remove a directory in a physical file system.

Table 1: Uniform Interface to Access Protocols

2.2 Logical Name Space

Like most Unix-like file systems, Jade presents a tree-structured naming hierarchy to the user. Unlike other file systems, each Jade file system is defined on a per-user basis. The result is a collection of small, per-user name spaces rather than a large system-wide name space. A Jade file name, rather than being global, has *scope* relative to a single logical name space. That is, every resolution of a file name is performed in the context of a specific user's name space.

A given logical name space is built on top of one or more physical file systems. Users choose the physical file systems they want to access, and glue these systems together to form their private logical file systems. Hence, it is possible to partition a given logical name space into multiple *domains*, each of which is implemented by a different physical file system. For example, Figure 1 illustrates a partitioning of a logical name space into domains.

Like most distributed file systems, Jade supports a *mount* operation that attaches a given physical file system to a logical name space. Unlike other systems, however, one physical file system may be mounted in many Jade file systems, each time in a different place. Because of autonomy, all the information necessary to mount one directory under another is maintained in the Jade file system; none of the underlying physical file systems is aware of the fact that it is participating in some user's logical file system.

Jade implements *skeleton directories* to maintain this mounting information and to keep track of the boundaries between the mounted file systems. We call it a skeleton directory because most of the files/directories within a given domain are maintained by some physical file system; only the roots of mounted file systems are maintained by Jade. Another way of saying this is that skeleton directories are superimposed over a collection of existing file hierarchies; much of the structure of the underlying hierarchies

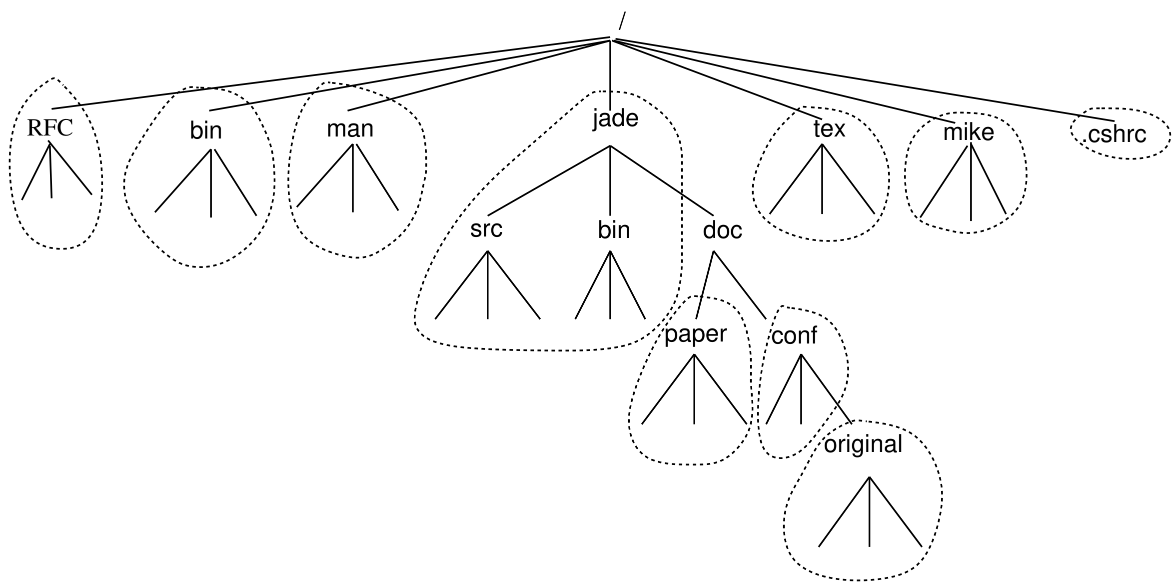

Figure 1: Partition of a Logical Name Space into Domains

remains visible to the user. For example, in Figure 1, the domain rooted at **/jade** represents one mounted file system, and we say that **/jade** is a skeleton directory.

Each skeleton directory defines of an ordered list of references. Each reference identifies a point in a mounted file system, and is given by the 4-tuple:

<Protocol, Server, Handle, Token>

Protocol identifies the protocol used to access the mounted file system, e.g., NFS, AFS, UFS, or FTP. **Server** specifies the host that provides services to access the mounted file system. **Handle** is the descriptor used by the server to identify the root of the mounted file system, for example **/usr**. **Token** provides authentication information used to access the mounted file system. The exact information specified in a reference varies according to the access protocol. For example, the host address is used to specify **Server** if the access protocol is NFS, while the cell name is used for AFS.

Jade maintains a per-user token list. Each **Token** in the list is assigned a generic name by the user and consists of a principal (e.g., login name) and an authentication key (e.g., password). For example, a user can define a token named **nobody** with the principal given by **anonymous** and the authentication key given by **ident**. Thus, a reference associated with the pathname **/RFC** in the name space illustrated in Figure 1 is given by

<FTP, nic.ddn.mil, RFC, nobody>

In order to access the file system **nic.ddn.mil:RFC**, Jade issues the user name **anonymous** and the password **ident** to the access protocol (FTP in this example).

To illustrate how various file systems are mounted into a Jade name space, consider the following sequence of examples, all relative to Figure 1. For simplicity, we ignore the **Token** component of the reference in the following discussion.

2.2.1 Simple Mounts

In the simplest case, the skeleton directory **/jade** contains the following reference:

{<NFS, meg.cs.arizona.edu, /usr/john/jade>}

where **meg.cs.arizona.edu** (**meg** for short) is the host name of the server, **/usr/john/jade** is the root of the mounted file system, and **NFS** indicates the NFS protocol used to access files on **meg**.

Similarly, the skeleton directory **/jade/doc/paper** contains the following reference:

{<AFS, cs.arizona.edu, user.john:/afs/az/usr/john/paper>}

where **AFS** indicates that the AFS protocol is used to access files in this domain, **cs.arizona.edu** is the cell name, and **user.john** is the volume name of the mounted file system and **/afs/az/usr/john/paper** is the name of the root.

2.2.2 Mounting Logical Directories

A skeleton directory might refer to another Jade file system rather than a physical file system. For example, the skeleton directory **/mike** refers to another Jade file system named **mike@cs.arizona.edu**. The reference is given by

{<JNP, mike@cs.arizona.edu, /database>}

where **JNP** (Jade Naming Protocol) [Rao91] is the protocol used to access a Jade name space, and **mike@cs.arizona.edu** identifies the logical name space belonging to user **mike** on host **cs.arizona.edu**.

To continue this example, suppose Mike's name space contains a reference to another Jade name space, as illustrated in Figure 2. In this example, the pathname **/mike/dfs.bib** in John's name space and the pathname **/database/dfs.bib** in Mike's name space refer to the same file. Furthermore, because Mike's name space mounts the subtree **/bib/journal** of Bob's name space under the skeleton directory **/database/ieee**,

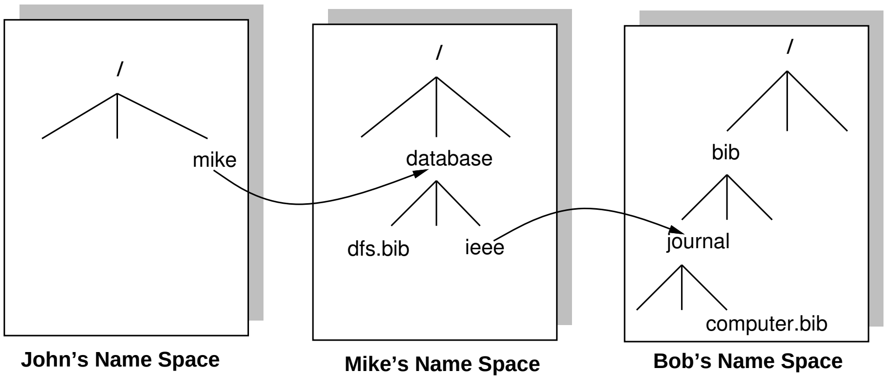

Figure 2: Mounting Other Name Spaces

the pathname **/mike/ieee/computer.bib** in John's name space refers to the file specified by the pathname **/bib/journal/computer.bib** in Bob's name space.

By mounting logical name spaces, Jade file systems are *linked* to each other to form a loosely coupled confederation. Figure 3 depicts the relationship between one logical name space and a collection of other logical and physical name spaces. The idea is that a pool of physical file systems are available in the internet, each of which is viewed as a *building block* with a uniform interface. Users construct their name spaces with these building blocks according to their own preferences. One logical name space, however, may point to several physical file systems as well as other logical name spaces. This indirect naming can be of arbitrary depth, but it is completely transparent to the user. The relationship among all logical name spaces is, however, arbitrary and voluntary without central authorities [Mich87], specific configurations [Cher89], or any kind of naming conventions [Zaya88, Ever90]. Because of autonomy, this property is extremely important to scale well in the internet.

2.2.3 Multiple Mounts

A skeleton directory might have more than one reference. We call this a *multiple mount*. For example, there are three references associated with the skeleton directory **/bin**:

```
{
    <UFS, jag.cs.arizona.edu, /usr/john/bin>
    <NFS, meg.cs.arizona.edu, /usr/john/bin>
    <NFS, meg.cs.arizona.edu, /usr/bin>
}
```

As another example, the references of the skeleton directory **/RFC** are

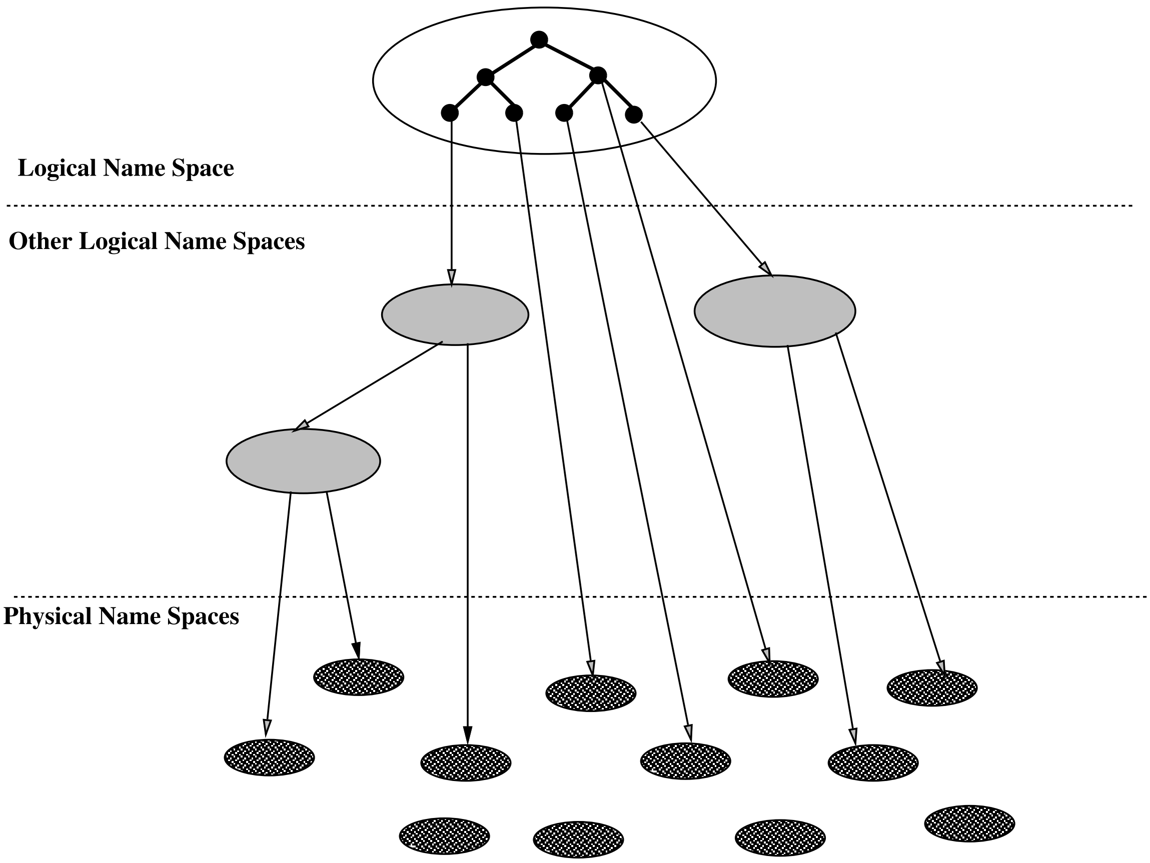

Figure 3: Relationship Among Logical and Physical Name Spaces

{
<NFS, meg.cs.ariozna.edu, /usr/john/RFC>
<FTP, nic.ddn.mil, RFC>
}

From the user's perspective, the entries in a skeleton directory that corresponds to a multiple mounts are computed by *combining* the entries in the mounted file systems. The combination operation is, however, applied only to entries under the skeleton directory; it is not recursively applied to the entire subtrees of all the mounted file systems. That is, the entries of the skeleton directory with the multiple mount are indeed the union of those on different mounted file systems. Entries of a directory under this skeleton directory, however, include *only* entries on the physical file system where this directory is located.

The mounted file system (either a physical file system or a logical file system) can have files with names that already exist as skeleton directories in the original name space. Also, files from different file systems mounted on one skeleton directory (multiple mount) can have the same name. Jade uses two rules to resolve name conflicts. First, names of local skeleton directories have precedence over names from mounted file systems. Second, the order of the list of references associated with the directory is used to resolve conflicts among different mounted file systems. Thus, files from the mounted file system appearing in the front of the list of references have preference over those appearing in the back of the list.

Because a given skeleton directory may refer to more than one file system, the file system on which a new file/directory should be created becomes an issue. Note that a given Jade pathname refers to at most one logical directory in the Jade name space, or one physical file/directory in a physical file system. In Jade, a new file/directory is created on the same physical file system where its parent directory is located. If the parent directory is a physical directory, the problem is trivial. If the parent directory is a skeleton directory (called the skeleton-parent directory), however, the first physical file system to which this directory refers is used. Note that the first physical file system is not necessarily referenced directly in the reference list associated with the skeleton-parent directory. It may be necessary to consult several logical name spaces to locate the desired file system. If the skeleton-parent directory points to no physical file systems, the operation of creating a new file fails. Finally, skeleton directories are created by the mount operation, which is completely separated from regular file/directory creation.

2.2.4 Null Mounts

In Jade, it is possible to mount no file systems into a skeleton directory; e.g., **/jade/doc** contains an empty reference list and corresponds to no physical file systems. Notice, however, that **/jade/doc** has two children;

i.e., when listing the contents of **/jade/doc**, the user sees both **paper** and **conf**. We call the skeleton directory **/jade/doc** a *purely logical* directory. Another example of a purely logical directory is the root node (**/**).

The subtle point illustrated by null mounts is that skeleton directories may both point to directories in other file systems and contain their own skeleton children. In the implementation, skeleton directories contain only the list of references to other file systems; they do not "point to" their own children. A hash table is used to locate a particular skeleton directory, given its name in the logical hierarchy. The issue of skeleton directories serving the dual role of symbolic link and directory is discussed further in Section 4.2.

2.3 Access to Remote Files

Jade employs whole file caching of remote files. When a file is opened, Jade checks the cache for the presence of a valid cached copy. If such a copy exists, the cached copy is opened and used. Otherwise, the Fetch operation is invoked to get an up-to-date copy from the original file in a physical file system. Read and write operations on an open file are directed to the cached copy. If a cached file is modified, it is stored back using the Restore operation to the physical file system when the file is closed.

Furthermore, Jade allows the user to choose any one of the underlying physical file systems as the cache server. Of course, the cache server is usually located nearby. Notice that the local disk of the workstation is considered as one of the physical file systems for Jade, and can be chosen as the cache server. The major advantage of this design is that it allows use of the logical file system even without a local disk for caching. It also provides flexibility in that users can dynamically switch the cache server to other physical file systems. Dynamically changing the cache server is particularly useful when the current cache server's storage is not big enough for remote files or is temporarily unreachable.

Finally, the Fetch and Restore operations defined by the uniform interface invoke the access protocol of the physical file system where the file is located. After the file is cached on the cache server, however, the access protocol provided by the cache server is used to open, read, write, seek, and close the cached copy on the cache server. In other words, cached copies on the cache server are accessed through the interface of the workstation operating system rather than the uniform interface defined in Section 2.1. Figure 4 schematically depicts a file that is located on a physical file system, cached on the cache server with Fetch/Restore operations, and accessed through a workstation using Read/Write operations.

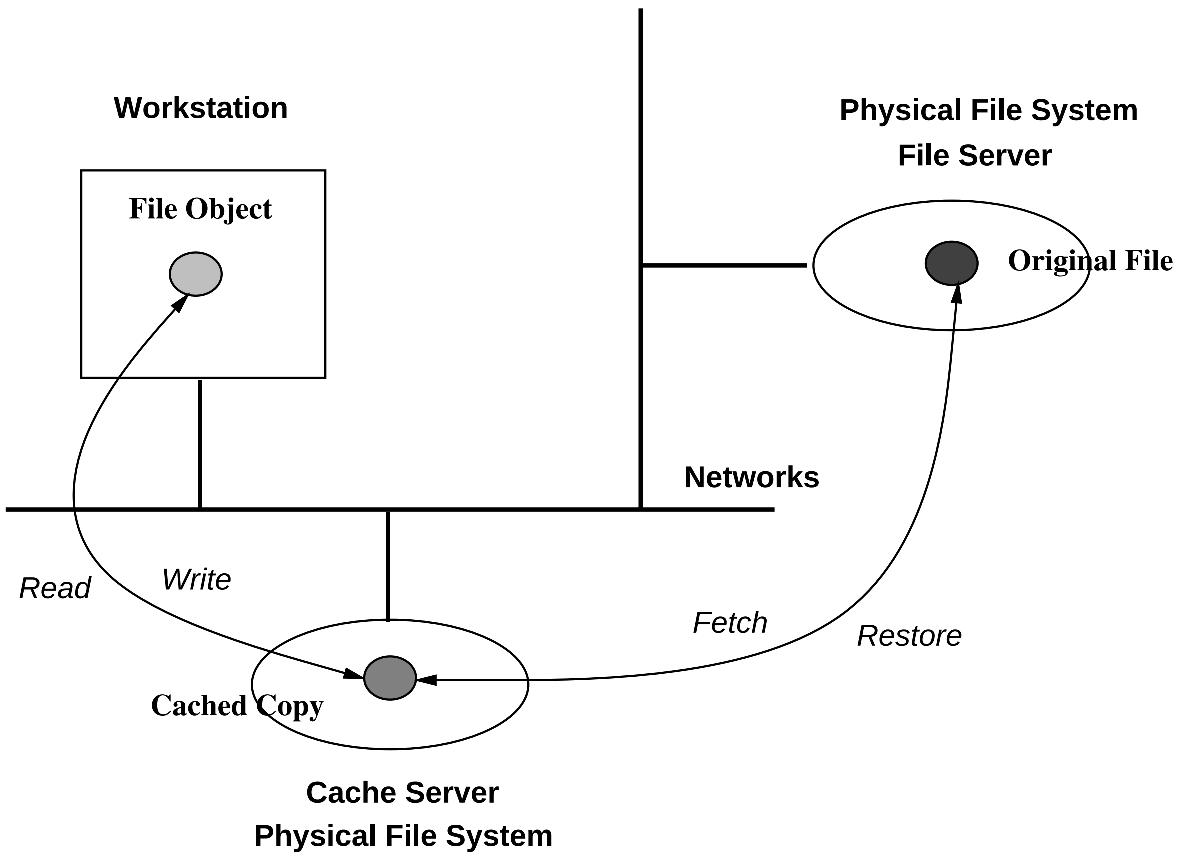

Figure 4: File Access

3 Performance

This section evaluates the Jade file system by outlining the prototype implementation and discussing its performance.

3.1 Prototype Implementation

Jade provides a logical layer between existing file systems and their users. Conceptually, it consists of two major pieces: a Name Space Manager and an Access Manager. The Name Space Manager provides a directory service that maps the logical file name provided by the user into a file reference. The Name Space Manager is called when the file is opened. Given a file reference, the Access Manager supports file access by caching entire files on a nearby physical file system. A key aspect of this design is that the directory service is completely separated from the file access service, both in functionality and implementation. The former is provided by the Name Space Manager, and the latter is supported by the Access Manager. Both the Name Space Manager and the Access Manager are built on top of the uniform interface described in Section 2.1, depending indirectly on the underlying file access protocols.

The prototype is implemented on top of the Sun OS 4.1 operating system; its structure is illustrated in Figure 5. Applications access Jade through a shared library. By dynamically linking the Jade library, most existing software (*ed*, *cc*, *find*, etc.) is able to transparently access Jade without modification or re-compilation. The Name Space Manager and the Access Manager are each implemented by a separate Unix process. The Jade library communicates with these two processes using Sun RPC. Finally, Jade implements each of the access protocols (i.e., UFS, NFS, and FTP) as an *agent*. Each agent implements the client part of one of the access protocols, and exports the uniform interface defined in Section 2.1.

Note that the prototype runs at user-level and requires no kernel modifications. Compared with the kernel-approach implementation used by the Andrew File System and the Network File System, the user-level approach has the following advantages. First, it is easy to experiment and to examine different design options. Second, debugging user-level servers is much easier than kernel-level mechanisms because the servers are ordinary applications and the standard debugging tools can be used. Third, portability among heterogeneous operating systems is easier. A potential disadvantage of this approach, however, is that performance will be degraded by the user-level approach. The next section addresses this issue in more detail.

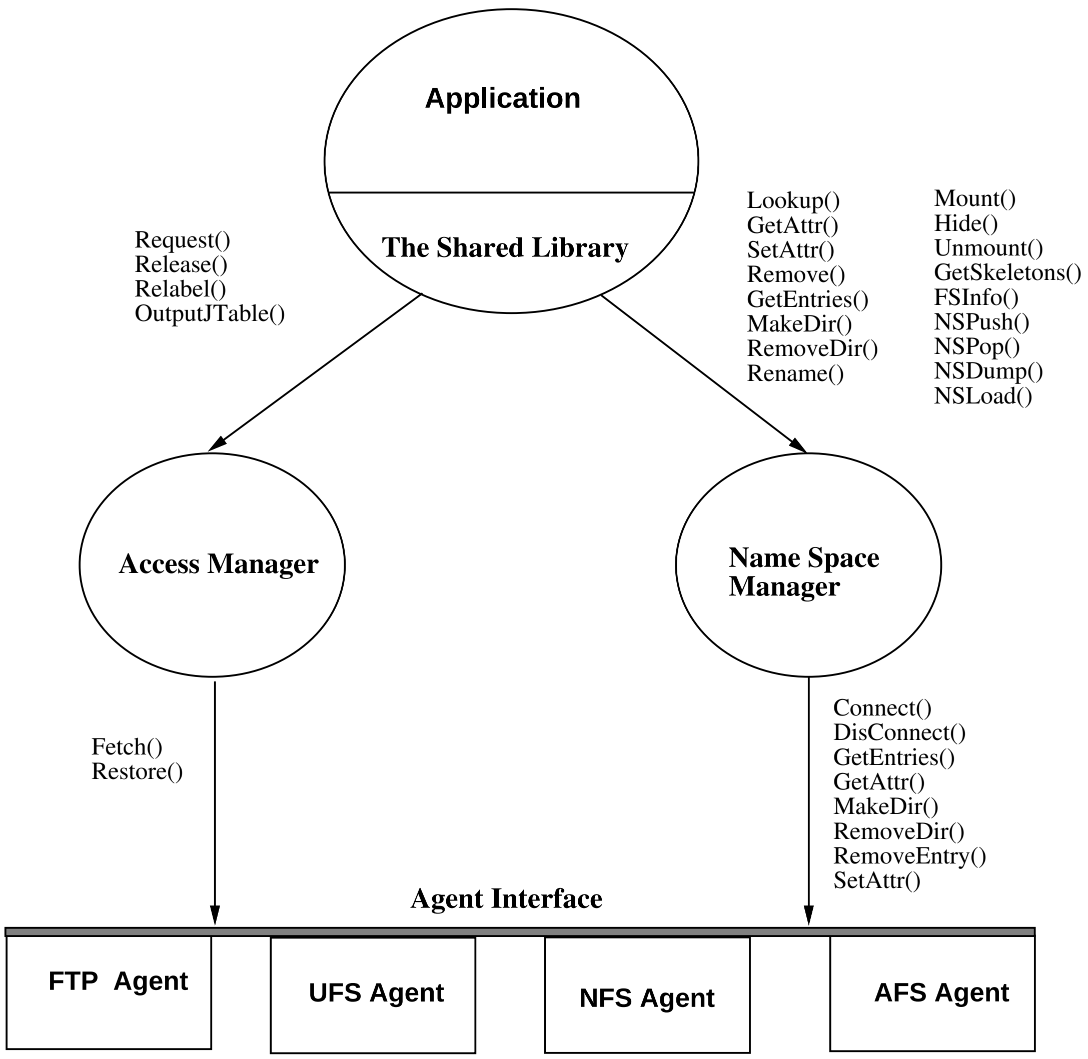

Figure 5: Implementation Structure

3.2 Experiments

In order to evaluate Jade's design, we have measured the performance of the prototype implementation using the Andrew Benchmark [Howa88]. The input to the benchmark is a read-only source subtree consisting of about 70 files. These files are the source codes of an application program and total about 200 kilobytes in size. The benchmark includes five distinct phases as listed in Table 2.

MakeDir	Constructs a target subtree that is identical in structure to the source subtree.
Copy	Copies every file from the source subtree to the target subtree.
ScanDir	Recursively traverses the target subtree and examines the status of every file in it. It does not actually read the contents of any file.
ReadAll	Scans every byte of every file in the target subtree once.
Make	Compiles and links all the files in the target subtree.

Table 2: Five Phases in Andrew Benchmark

Two cases are tested: a local area network and the internet. For each case, we compare the performance of Jade and the NFS.[1] For the first case, the file sever is located on a local area network, where a 10 Mbps Ethernet connects the file server and the client workstation. For the internet test, the file server is located at Purdue University and the client workstation is at the University of Arizona. There are 13 gateways between the file server and the client workstation, and the communication channels between them range from 10 Mbps Ethernets to 1.544 Mbps T1 connections. The client workstation, where Jade and the Andrew Benchmark running, is a Sun 4/60 workstation with 16 Mbytes main memory and 320 Mbytes disk dedicated for files caching by the Access Manager. Both file servers provide the NFS protocol for file access.

The performance results of both tests are given in Table 3. In the LAN case, the Access Manager is running on the host where the file server is located, and therefore there is no need to cache files whenever accessing them. Jade exhibits a 36% slowdown relative to NFS. We attribute this to the cost of the user-level implementation. For example, each **open** call needs to consult the Name Space Manager to resolve pathnames; the Access Manager is not consulted because the files are located on the same host as the Access Manager. Since the Name Space Manager is running as a separate process, there are six user-kernel boundary crossings; NFS only requires two crossings for this test. This result (36% slowdown) is similar to the result from the Pseudo-File-System [Welc89]. The Pseudo-File-System provides access to NFS file servers from Sprite workstations and experiences a 33-41% slowdown when running the Andrew Benchmark.

[1] In this section, we use "NFS" to refer to the complete file system, and we say "NFS protocol" when referring to the access protocol used by the Network File System.

	LAN		Internet	
	NFS (sec)	Jade (sec)	NFS (sec)	Jade (sec)
MakeDir	3	3	23	23
Copy	20	23	299	536
ScanDir	31	52	115	127
ReadAll	50	84	120	139
Make	98	113	568	344
Total	202	275	1125	1169

Table 3: Performance Results

In the internet case, the overall performance of Jade is almost identical to that of NFS, with only a 4% slowdown. The general observation is that the cost to access the internet is so high that the penalty of the user-level implementation is insignificant. Entire file caching is another interesting issue. Jade takes advantage of the fact that the cached copies can be reused in the latter operations, and therefore in the last phase of the test, **Make**, Jade's time dramatically drops to 61% of the NFS time.

The major drawback of this access pattern is that the cost of copying files is extremely high. Consider three alternative ways to copy a file *A* to a file *B*, assuming *A* and *B* are located in the same physical file system. With the basic case, the file *A* is *fetched* from its source to the Access Manager to generate a cached copy of *A*, the cached copy of *A* is copied to another cached copy in the Access Manager which is then used as the cached copy of the file *B*, and the cached copy of *B* is stored back to its destination. The total cost is three copy operations. With the optimized case, Jade provides a new function **Relabel** to let users change the reference associated with the cached copy from its source to a new sink, and therefore the copy from the cached copy of *A* to the cached copy of *B* can be omitted. However two copy operations are still required, and the performance of the **Copy** phase of the Andrew Benchmark exhibits a 79% slowdown compared with the NFS case in which only one copy operation is performed. The ideal way to solve this problem is to have the access protocol support the new function `copy`, and therefore the cost of copying files is comparable to the cost of *renaming* files.

For the **ScanDir** phase, the performance of Jade in the LAN case is 52 seconds, while it is 31 seconds for NSF. In the internet case, the former is 127 seconds and the latter is 115 seconds. The lesson we learn from this phase is obvious: network latency is a significant factor in an internet. Avoiding unnecessary network messages is crucial in performance improvement. For example, the NFS protocol supports the function readdir to list entries under a given directory. However, it only returns a file name for each entry. In order to obtain full information about a directory for each entry, it requires one extra function call, lookup, to retrieve the file's attributes. In the LAN case, where network latency is not a issue, this overhead is insignificant. In the internet, where the network latency is much higher, the cost becomes visible and even serious.

We have extended the function readdir to return attribute information in addition to the name of each entry in a directory. The performance of this phase in the internet case then improves to 72 seconds, which is 43% faster than the Jade figure presented in Table 3 and 38% faster than NFS. Notice that the speedup percentage will increase as a function of directory size because the fixed cost of reading a directory is amortized over more directory entries.

For the **ReadAll** phase, the performance of Jade in the LAN case is 84 seconds and it is 50 seconds for NSF. For the internet case, the former is 139 seconds and the latter is 120 seconds. However, when running this phase in the internet case, NFS, which uses page access, has a similar performance. Jade drops to 65 seconds, which is 46% faster than NFS. This is because Jade takes advantage of the fact that cached files on the Access Manager can be reused. Again, the **Make** phase illustrates that caching entire files is essential for good performance in the internet. In the LAN case, Jade takes 113 seconds and NFS takes 98 seconds, while in the internet case, Jade's time drops dramatically to 61% of the NFS time (344 seconds versus 568 seconds). This result is extremely important in an engineering environment where the majority of file access is editing and compiling a small set of files [Floy86].

4 Discussion

Jade's main goal is to provide scalable access to internet files. This section identifies and justifies the design choices we made trying achieving this goal. It also contrasts our approach with related work.

4.1 Global vs. Per-User Name Space

Defining the file system on a per-user basis seems to be well justified. The activity of accessing files by a single user tends to be isolated from other users, and focused on a small working set of directories [Cabr88, Floy86, Shel86]. Satyanarayanan [Saty89] has pointed out that in a research or academic environment, most files are read and written by a single user. When users share a file, it is usually the case that only one of them modifies it. This implies that file references outside the user's private name space are relatively infrequent.

The alternative is a single, global naming hierarchy. The advantage of a global name space is that it supports a coherent view among users and hence promotes resource sharing. Most early experience with global name spaces in a distributed environment has been in the context naming hosts and mailboxes; for example, Grapevine [Birr82], Lampson's Global Name Space [Lamp86], the Domain Name Service [Mock87], and X.500 [ISO88]. The general criticism of these systems is that they are too heavy-weight to support the frequent updates and lookups required by a file system.

In the context of file systems, a global name space often implies the existence of *central control*, and central control is not amenable to scalable systems. For example, the Amoeba File System [Tane90] uses a central directory server to map a string name of an object into its capability. This central server would be a performance bottleneck when the system becomes large scale. As another example, the Sprite File System employs a prefix table to map pathname prefixes to file servers in a distributed environment and uses *broadcasting* to locate the file server whenever prefix matching fails. As with central control, broadcasting is unrealistic in a an internet environment. As a final example, the Network File System requires all the hosts to mount each other's file systems in order to create a global name space. The number of mount points in the system, therefore, is proportional to the square of the number of hosts in the environment, limiting its scalability.

Recently, there have been efforts to extend a distributed file system that uses a global name space to the internet. One example is the Andrew File System. The original design [Howa88] considered only one aspect of scalability—size. An extended design, called the Cellular Andrew Environment [Zaya88], considers a wide-area environment and allows a collection of sites to cooperatively establish a global name space among these sites. In order to construct such a global name space, however, each site must adopt the Andrew File System, making autonomy the major problem of this design. It has proven difficult to persuade each autonomous site to give up the file systems currently running, and switch to the Andrew File System.

Another example of a global name space in a wide-area network is Cheriton and Mann's *decentralized naming* scheme. With their scheme, the naming hierarchy is partitioned into global directories, regional directories, and local directories, with different mechanisms used at each level. In reality, however, the availability of multicasting mechanisms in the internet, which are used to locate name servers, is the major limitation of this design. Even when multicast mechanisms are available in the internet, this design does not solve other drawbacks of the global name space approach, including the difficulty of searching for files due to long pathnames, the lack of flexibility to tailor the name space for individual needs, and the infringement on site autonomy.

In contrast to these examples of global name spaces that introduce some level of decentralization to achieve scalability, Jade *completely* decentralizes the construction and maintenance of name spaces by placing control in the hands of individual users rather than system administrator. This seems reasonable since the scope and complexity of a per-user name space is much less than a that of a global name space. This is because although the number of available file systems in an internet is huge and growing rapidly, the number of file systems an individual user wants to access at one time remains small and relatively stable.

The bottom line is that the per-user name space trades the burden on administrators of maintaining a global name space for the burden on users of organizing private name spaces. As just pointed out, maintaining a consistent and coherent global name space in a large distributed system is not a trivial task. On the other hand, organizing a private name space is much easier. The scope of the private name space is both small and relatively static. A default private name space for novice users, which includes the directory for binaries and the user's home directory, can be automatically generated from user password files (e.g., **passwd** file in Unix). Expert users can then tailor their own file systems by mounting the desired file systems into their logical name spaces. The user must know where a physical file system is located to be able to mount it on his or her logical file system, but once the file system is mounted, the user can use the logical file system in a network transparent way.

Finally, there are other distributed file systems that, like Jade, provide mechanisms to let users construct their own name spaces. Examples include Tilde [Come86], QuickSilver [Cabr88], and Plan 9 [Pres91]. QuickSilver and Plan 9 do consider systems in large scale, but only in terms of size and wide area. Jade surpasses these systems in the ability to accommodate heterogeneity, allow for customization, and support interactions between name spaces. Generally, none of these three systems allows a name space to be mounted into another name space, and they are all limited to a single access protocol.

4.2 Skeleton Directories

Skeleton directories play a key role in Jade. They differ from similar mechanisms found in other distributed file systems in the following ways.

First, consider skeleton directories at the implementation level. Skeleton directories are similar to *prefix tables* and *remote links* used by the Sprite File System [Welc86], *mounting tables* and *mount points* used by the Network File System, and *volumes* used by the Andrew File System [Side86]. The fundamental difference comes from where and how the mounting information is maintained. Andrew embeds all mounting information in volumes that are maintained by physical file systems. In Sprite, the remote link is maintained by the physical file system and used as a marker of the boundary; the prefix table is located on the client site, but serves as a naming cache only. Thus, both Andrew and Sprite embed mounting information in the data stored in the file server, which Jade cannot do, not only because of the autonomy restriction, but also because each user may mount the file system in a different place in her or his own logical name space.

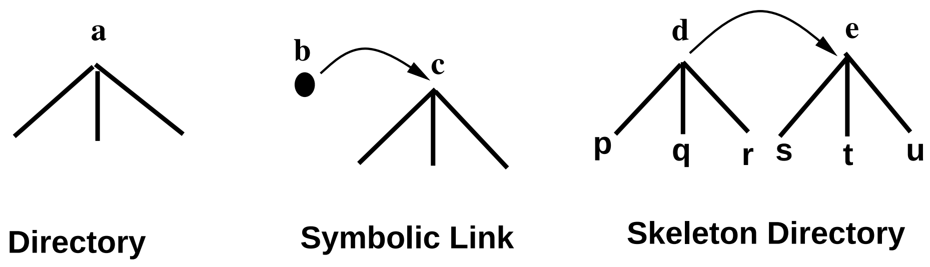

Figure 6: Directories, Symbolic Links, and Skeleton Directories

The Network File System separates the mounting information into two parts: The mounting table directs a path name to the appropriate file server, and mount points relate a local directory to the root of the mounted file system. The former is kept in the client site, while the latter is maintained by individual physical file systems. Jade, on the other hand, realizes the mounting relation as the skeleton directory and maintains it only at the client. More precisely, when one physical file system is mounted under another, the latter system contains no information pointing to the former. All information needed to mount one directory under another is maintained at the client.

Second, consider skeleton directories at an abstract level, and in particular, how they compare to symbolic links found in most Unix-like file systems. Like a skeleton directory, a symbolic link redirects a path from one subtree to another. Unlike a skeleton directory, a symbolic link can only point to at most one subtree. Moreover, symbolic links are always leaves in a naming tree. For example, suppose **/a/b** is a symbolic link to the directory **/a/c**. A new file **/a/b/foo** would be created as an entry in the directory **/a/c** instead of **/a/b**. A skeleton directory, on the other hand, may have a set of skeleton directories under it.

Finally, skeleton directories are a generalization of symbolic links and directories. Figure 6 sketches a directory, a symbolic link, and a skeleton directory. In the following discussion, the term *local entries* of a directory is used to refer to entries that are maintained by the directory itself. For example, entries **p, q,** and **r** are local entries of the directory **d**. The directory **a** contains only local entries. The symbolic link **b** refers to a node **c**, and thus, entries under **b** are the same as those under **c**. The skeleton directory **d** not only contains local entries but also refers to another node **e**. Therefore, entries under **d** are the union of its local entries and entries under **e**—that is, **p, q, r, s, t**, and **u**. In general, the node referred to by a skeleton directory can be either a symbolic link, a directory, or another skeleton directory.

4.3 Logical Name Spaces

The logical name spaces found in Jade provide an important new mechanism for organizing related files, much in the same way as directories do in most existing file systems. This is because users can define Jade name spaces with a wide spectrum of granularity. At one end of the spectrum, a user may define more than one name space in order to handle different tasks, e.g., one for teaching, several for different research projects, one for administration, and so on. At the other end of the spectrum, it is possible to define a logical name space for a software project that is shared by users working on the project. A large software project may even have a set of name spaces, each of which represents a view for one particular software/hardware configuration, e.g., one for shared codes, one for the Sparc architecture, one for Sequent machines, another for Sun OS 4.1, and so on. Furthermore, Jade treats logical name spaces as computable objects in that it allows users to construct their views of file systems by composing multiple name spaces. With this feature, Jade is able to support the same functionality as the *viewpath mechanism*, which is used by configuration tools (e.g., make) and software development environments. Since this function is embedded into the file system rather than individual tools, all commands are able to take advantage of it. The paper [Rao91] illustrates how Jade's logical name spaces can be used to provide a powerful software development platform, one that is especially suited for wide area software environments.

On a related point, Jade performs pathname resolution in the context of a specific name space. When running a program, the name space of the user who invoked the program (called the *invoker*) is used by default to resolve names. However, it is possible to associate a particular name space with a program. This attached name space is used for name resolution when the program is executed. This feature is similar to the function setuid provided by Unix: It changes the privilege of a process from the program invoker to the program owner. For example, when running a text processing application, the application can use its name space rather than the invoker's name space to resolve font file names. Furthermore, by overlapping the application's name space and the invoker's name space, it is possible to resolve font file name in the former while input file names in the latter.

4.4 Multiple vs. Union Mounts

Korn and Krell's 3-D file system [Korn90] and Sun's Translucent File Service (TFS) [Hend90] advocate a *union mount* mechanism that is different from the multiple mount provided by Jade. With the union mount, *entire* subtrees from different mounted file systems are merged. With multiple mounts, on the other hand, entries of the skeleton directory are the union of those on different mounted file systems; entries of a directory under this skeleton directory include *only* entries on the physical file system where this directory is located.

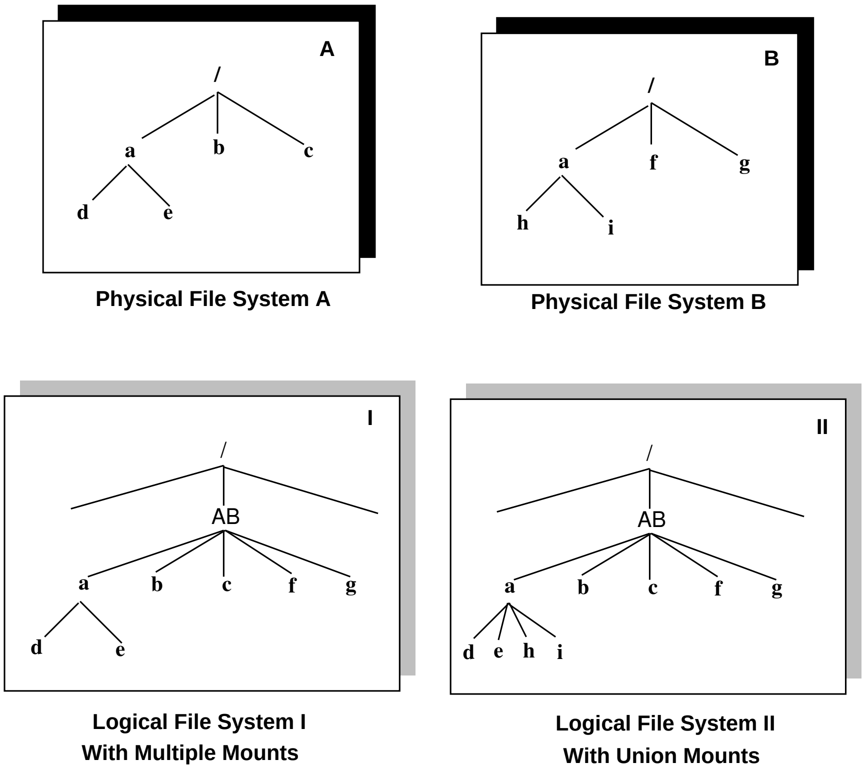

Figure 7: Comparison of Multiple Mounts and Union Mounts

In order to compare multiple mounts and union mounts, consider two physical file systems **A** and **B** and two logical file systems **I** and **II** as illustrated in Figure 7. Both **I** and **II** mount **A** and **B** on the path **/AB**, while **I** uses multiple mounts and **II** uses union mounts. The directories named **/AB** on **I** and **II** have the same entries. However, the directory named **/AB/a** on **I** has entries only from the directory **/a** on **A** (i.e., **d** and **e**), while the directory **/AB/a** on **II** has entries that are the union of those from the directory **/a** on **A** and those from the directory **/a** on **B** (i.e., **d, e, h**, and **i**).

Multiple mounts can simulate union mounts. That is, by recursively applying multiple mounts, users can generate exactly the same view on mounted file systems as by using union mounts. For example, by mounting both **A:/a** and **B:/a** on the path **/AB/a**, **I** has the exact same view as **II**. On the other hand, by mounting only **A:/a** on the path **/AB/a**, **II** restricts the entries under **/AB/a** to only those from **A**, and therefore presents the same view as **I**.

There are two reasons why Jade supports multiple mounts rather than union mounts. First, it is more expensive to resolve a pathname in a name space with union mounts. This is because whenever it fails to resolve a name in one directory, the resolution process must *backtrack* to the original skeleton directory and try other mounted file systems. This backtracking process becomes even more complicated in the general case where nodes in mounted file systems can also be skeleton directories pointing to multiple file systems.

Second, with multiple mounts, a pathname refers to at most one directory—either a skeleton directory in a logical name space or a physical directory in a physical file system. With union mounts, on the other hand, a given pathname may refer to more than one physical directory in different physical file systems. For example, the path **/AB/a** in **II** refers to two directories, **/a** in **A** and **/a** in **B**. The property of one-to-one mapping from pathnames to nodes in a naming tree is preserved in a name space with multiple mounts, but not in a name space with union mounts. Many problems arise without this property. For example, it is difficult to maintain directory attributes in the union mount case; it is not clear where the attributes are recorded.

4.5 Pathname Resolution

Jade implements a logical name space by maintaining skeleton directories that keep track of the boundaries between the underlying file systems. When resolving a given pathname, Jade first locates the proper skeleton directory, and then resolves the rest of the pathname by consulting the underlying file systems referred to by references stored in the skeleton directory.

Because a given skeleton directory may have more than one reference, as well as references to other logical file systems, the procedure to resolve the rest of the pathname is more complicated than those used by other distributed file systems. Three issues need to be considered. The first involves the simple matter of resolving the pathname on a physical file system. The key to this issue is achieving acceptable performance. The second issue involves resolving a name relative to more than one logical name space. Since Jade allows pathnames across logical name space boundaries, the searching procedure may invoke a *sequence* of logical name spaces before reaching the physical file system that is able to complete the resolution process. The last issue concerns multiple mounts, in which case it may be necessary to try several possibilities before successfully resolving the given name. The resolution algorithm is discussed below, focusing on the tradeoff among different algorithms. A complete description of the algorithm is given in [Rao91].

4.5.1 Local vs. Remote Resolution

When resolving the remaining path on the physical file system, Jade fetches each directory from the remote host and searches it on the local host. This method is called *local pathname resolution*. With the alternative, called *remote pathname resolution*, the pathname is forwarded to each remote host, which opens and searches its own directories.

Like Locus and Andrew, Jade adopts the local resolution method and caches directories that is has fetched from remote hosts. Our decision to use local resolution was based on experiments [Shel86, Howa88] demonstrating that the activity of most users is confined to a small, slowly changing subset of the entire name space hierarchy. Thus, a directory cache has a high hit ratio, and much of the network traffic for importing directory entries from remote file systems is avoided. When network latency becomes significant, as in an internet, avoiding unnecessary network messages between clients and file servers is crucial to achieve acceptable performance.

4.5.2 Iterative vs. Recursive Resolution

Being able to mount logical file systems means that a sequence of file systems may need to be searched to resolve a given pathname. In Jade, the original name space *iteratively* calls each name space in the sequence; the alternative is for the name spaces to call each other *recursively* [Terr85]. For example, suppose the original name space needs to contact name spaces **A**, **B** and **C**, in that order. In the iterative approach, the original name space first calls **A**, then **B**, and finally **C**. In the recursive approach, the original name space calls **A**, which calls **B**, which finally calls **C**.

The recursive method has the advantage of completely hiding forward mounts from the original name space. Therefore, the procedure for handling logical file system mounting is treated exactly the same as for physical file system mounting. This simplifies the interface design. However, this method is very expensive because it requires each logical name space in the calling sequence to collect directory entries before answering the query. Moreover, because of its recursive nature, the original name space has no control over the whole resolution activity, making the detection of loops in the mounting sequence more difficult.

The iterative method exposes skeleton directories; i.e., physical name spaces return real directories and logical name spaces return skeleton directories. Although it complicates the interface, this makes it easy to detect loops in the mounting graph. Another advantage is that it simplies authentication control. Consider a sequence of name spaces **A**, **B**, and **C**, where **C** is the physical file system where the desired files are located. With the iterative method, because the original name space maintains the inquirer's authentication information on a per-host basis, it can issue the proper authentication to **C**. The recursive method needs an extra pair of messages between **B** and the original name space to get the authentication information before **B** can query **C** on behalf of the original name space. When mounting relationships become even more complicated (e.g., multiple mounts), this overhead is even worse.

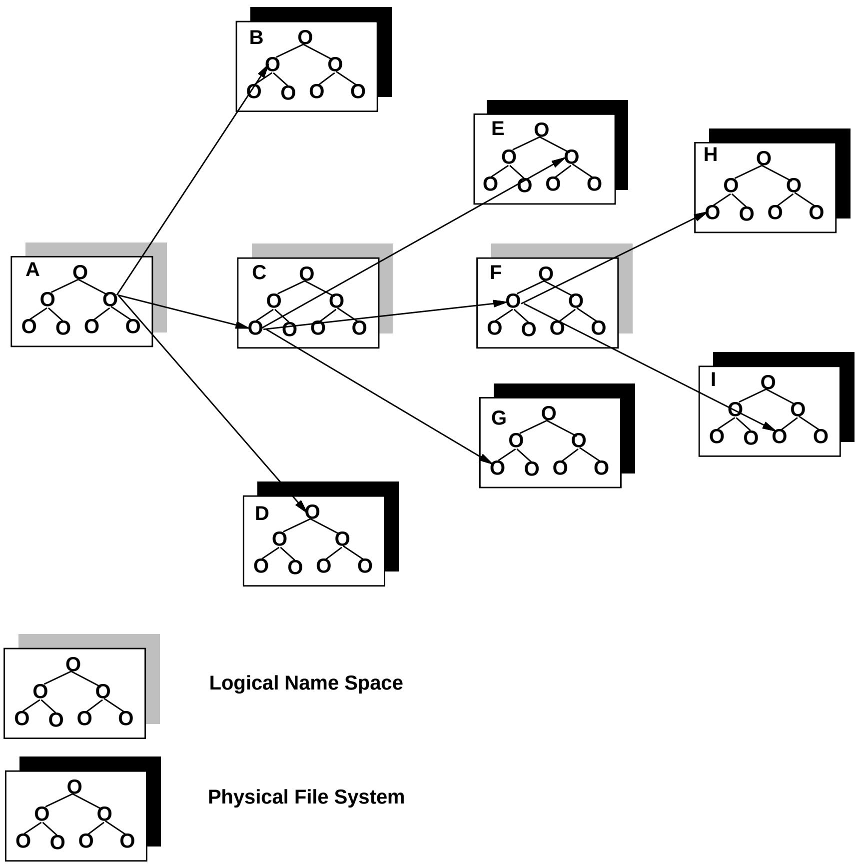

Figure 8: Mounting Graph

4.5.3 Handling Multiple Mounts

Jade allows more than one file system to be mounted on a single skeleton directory. The mounted file systems can be either physical file systems or other Jade file systems. In the latter case, a sequence of nodes in different logical name spaces may be consulted before proper physical file systems are located. However, multiple mounts may also occur in name spaces within this sequence. Hence, among these invoked name spaces, there is a directed graph that describes the mounting relationships. Figure 8 illustrates a directory of the name space **A** and all name spaces referenced by this directory: Name spaces **A**, **C**, and **F** are logical, and the name spaces of **B**, **D**, **E**, **G**, **H**, and **I** are physical. Arrows represent mounting relationships among name spaces. In this example, a skeleton directory in the name space **A** multiply mounts nodes on name spaces **B**, **C**, and **D**.

The preference rules presented in Section 2 dictate that a depth-first search is the proper way to search name spaces in the direct graph. In this example, the sequence of invoked name spaces is **B**, **C**, **E**, **F**, **H**, **I**, **G**, and **D**. Since name spaces could be located anywhere in the internet, the search terminates whenever the desired name is found in one of the name spaces of this sequence in order to improve performance.

4.6 Caching Entire Files

The Cedar file system [Giff88] introduces the concept of caching entire files on a workstation's local disk. The Andrew file system has shown that in a large environment this approach, together with a call-back mechanism, is superior in performance to the page access pattern used by the Network File System. In Jade, caching entire files is essential to access internet files, as described in the previous section. There are two reasons for this. First, physical file systems are contacted only on file opens and closes and not on individual reads and writes. Second, the total network overhead in transmitting a file is lower when the file is sent in its entirety rather than in a series of requests and responses for individual pages.

Jade differs from the Cache Manager of the Andrew File System in two respects. First, Jade does not require a local disk for caching. Instead, it allows the user to choose any one of the underlying physical file systems as the cache server. Second, unlike the Andrew File System, Jade integrates a collection of heterogeneous file access protocols (i.e., NFS, AFS, FTP, and UFS) and supports a uniform interface among them.

There is one obvious problem with this approach: accessing very large files. Jade provides a partial solution for this problem by allowing the user to choose one of many physical file systems as the cache server rather than restricting the cache server to the local disk. When necessary, the cache can be dynamically migrated to a larger file system—including the system where the file is located—in order to access larger files.

4.7 Cache Consistency

Jade guarantees sequential write sharing, but not concurrent write sharing [Nels88]. It does this by retrieving the time a file was last modified from the underlying file system each time it caches or writes back a copy of the file. When re-opening a file that is already in its cache, Jade validates its copy by comparing this time with the current modify time of the file on the server. Notice that this approach does not require a global clock; all times are read on the file's home server.

Contacting the server to validate a cached copy of a file is clearly more expensive than depending on a callback mechanism, such as can be found in the Andrew file system, but Jade is not able to use callbacks because not all file access protocols support them. In fact, callbacks do not even make sense for a page-based access protocol. Regarding stronger consistency guarantees—e.g., concurrent write sharing—we do not believe such guarantees are practical in an internet environment, and certainly not under the autonomy constraint that Jade adheres to. For example, adapting the DEcorum File System's *token mechanism* [Kaza90] to Jade would require modifications to the underlying access protocols, and would be extremely expensive and difficult to maintain because of the high network latency and unreliable nature in an internet. We conclude, therefore, that stronger consistency requirements should be the responsibility of application programs.

4.8 Access Control

Jade does not implement its own authentication control mechanism. Instead, it relies on the underlying file systems to check the access rights whenever their files are accessed. This is because Jade is just an *agent* between the user and the file system; it does not have any special privileges. One problem with this decision is that a skeleton directory owned by one user is readable by other users; i.e., users cannot make skeleton directories unreachable for others. However, because we use the iterative search method, the user still needs to have access rights in the physical file system in order to list the contents of a directory on a physical file system.

Unix access control requires that a user have access rights for every component in a pathname in order to access the file. Most Unix-like file systems implement this behavior by checking permission component by component starting from the root, during pathname resolution. In Jade, however, a given pathname may cross multiple domains, each of which may be located anywhere in the internet or be temporarily unreachable. It would be very expensive to check permission in each domain. Instead, Jade locates the desired domain directly using a hash table, and permission checking is done only inside this domain. For example, if there is a domain rooted by the skeleton directory **/a/b/c** and a lookup request on the path **/a/b/c/d**, neither of the directories **/a** or **/a/b** is examined. The node **/a/b/c** is used as a starting point to resolve the remaining path. This means that any access controls in **/a** and **/a/b** will be ignored. Therefore, if access to a file is to be restricted, it must be restricted with the access controls at the domain where this file is located.

5 Concluding Remarks

Most distributed file systems fail to scale from local area networks to an internet. We identify four dimensions to scalability: size, wide area, heterogeneity, and autonomy. Because of size and wide area, techniques such as broadcasting, central control, and central resources, which are adopted by many other file systems, are not adequate for an internet file system. An internet file system must also support the notion of autonomy because an internet is made up by a collection of autonomous organizations. Finally, heterogeneity is the nature of an internet file system not only because of its size but also because of its autonomous property.

Our primary goal is to design a file system for the internet environment that is both scalable and practical. In order to achieve the goal of scalability, Jade is partitioned into a collection of per-user, autonomous, logical file systems, each of which consists of a set of physical file systems and a dedicated logical name space. With the per-user approach, Jade fully decentralizes the construction and maintenance of name spaces from system administrators to individual users. Instead of introducing a new file system, we focus on accommodating existing distributed file systems. Jade also generalizes logical and physical file systems. It allows one logical file system to be mounted into another logical file system, in the same way that a physical file system can be mounted into a logical file system.

Jade is practical in two respects. First, it provides complete autonomy. It is implemented, as well as installed, on client sites without any modification to the software or administration policies of the servers. Therefore, Jade is more practical than file systems built from scratch that require considerable modifications to each of the underlying file systems. Second, we paid careful attention to avoiding unnecessary network messages between clients and file servers in order to achieve acceptable performance. Experiments with a prototype implementation demonstrate that Jade has acceptable performance in both the local case and the internet case.

References

[Birr82] Birrell, A. D., Levin, R., Needham, R. M., and Schroeder, M. D. Grapevine: An exercise in distributed computing. *Communications of the ACM*, 25 (4):260–274, April 1982.

[Cabr88] Cabrera, L. F. and Wyllie, J. Quicksilver distributed file services: An architecture for horizontal growth. In *Proceedings of the 2nd IEEE Conference on Computer Workstations*, pages 23–37, Santa Clara, CA, March 1988.

[Cher89] Cheriton, D. R. and Mann, T. P. Decentralizing a global naming service for improved performance and fault tolerance. *ACM Transactions on Computer Systems*, 7(2):147–183, May 1989.

[Come86] Comer, D. and Murtagh, T. P. The Tilde file naming scheme. *IEEE Transactions on Software Engineering*, pages 509–514, 1986.

[Ever90] Everhart, C. F. Conventions for names in the service directory in the AFS Distributed File System. Technical report, Transarc Corporation, March 1990.

[Floy86] Floyd, R. Short-term file reference patterns in a Unix environment. Technical Report TR 177, Computer Science Department, The University of Rochester, March 1986.

[Giff88] Gifford, D. K., Needham, R. M., and Schroeder, M. D. The Cedar file system. *Communications of the ACM*, 31(3):288–299, March 1988.

[Hend90] Hendricks, D. A filesystem for software development. In *Proceedings of Summer Usenix*, June 1990.

[Howa88] Howard, J. H., Kazar, M. L., Menees, S. G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N., and West, M. J. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[ISO88] ISO, . Information processing systems: open systems interconnection—the directory—overview of concepts, models, and service. Technical Report Draft International Standard ISO 9594-1:1988(E), International Standards Organization, 1988.

[Kaza90] Kazar, M. L., Leverett, B. W., Anderson, O. T., Apostolides, V., Bottos, B. A., Chutani, S., Everhart, C. F., Mason, W. A., Tu, S., and Zayas, E. R. DEcorum file system architectural overview. In *Proceedings of Summer Usenix*, July 1990.

[Korn90] Korn, D. and Krell, E. A new dimension for the Unix file system. *Software—Practice and Experience*, 20(S1):S1/19 – S1/34, July 1990.

[Lamp86] Lampson, B. W. Designing a global name service. In *Proceedings of Fifth Symposium on the Principles of Distributed Computing*, pages 1–10, August 1986.

[Levy90] Levy, E. and Silberschatz, A. Distributed file systems: Concepts and examples. *ACM Computing Surveys*, 22(4):321–374, December 1990.

[Mich87] Michael Schwartz, J. Z. and Notkin, D. A name service for evolving heterogeneous systems. Technical Report 87-02-05, CS Dept., University of Washington, February 1987.

[Mock87] Mockapetris, P. Domain names—implementation and specification. Request For Comments 1035, USC Information Sciences Institute, Marina del Ray, Calif., November 1987.

[Nels88] Nelson, M. N., Welch, B. B., and Ousterhout, J. K. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[Post85] Postel, J. and Reynolds, J. File Transfer Protocol (FTP). Request For Comments 959, USC Information Sciences Institute, Marina del Ray, Calif., October 1985.

[Pres91] Presotto, D., Pike, R., Thompson, K., and Trickey, H. Plan 9, a distributed system. Technical report, AT&T Bell Laboratories, 1991.

[Rao91] Rao, H. C. *The Jade File System*. PhD thesis, University of Arizona, 1991.

[Sand85] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B. Design and implementation of the Sun Network File System. In *Proceedings of Summer Usenix*, pages 119–130, June 1985.

[Saty89] Satyanarayanan, M. Distributed file systems. In Mullender, S., editor, *Distributed Systems*, pages 149–188. ACM Press, 1989.

[Shel86] Sheltzer, A. B., Lindell, R., and Popek, G. J. Name service locality and cache design in a distributed operating system. In *Proc. 6th Int. Conf. on Distributed Computing Systems*, pages 515–523, Cambridge, Massachusetts, May 1986.

[Side86] Sidebotham, B. Volumes: The Andrew File System data structuring primitive. In *European Unix User Group Conference Proceedings*, 1986.

[Svob84] Svobodova, L. File servers for network-based distributed systems. *ACM Computing Surveys*, 16(4):353–398, December 1984.

[Tane90] Tanenbaum, A. S., van Renesse, R., van Staveren, H., Sharp, G. J., Mullender, S. J., Jensen, J., and van Rossum, G. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, December 1990.

[Terr85] Terry, D. B. *Distributed Name Servers: Naming and Caching in Large Distributed Computing Environments*. PhD thesis, Univerity of California, Berkeley, 1985. Available as UCB/CSD Technical report 85/228, and as Xerox PARC Technical report CSL-85-1.

[Welc86] Welch, B. B. and Ousterhout, J. K. Prefix tables: a simple mechanism for locating files in a distributed system. In *Proceedings of the 6th Conference on Distributed Computing Systems*, pages 184–189, May 1986.

[Welc89] Welch, B. B. and Ousterhout, J. K. Pseudo-File-Systems. Technical Report UCB/CSD 89/499, University of California Berkeley, Berkeley, Calif., 1989.

[Zaya88] Zayas, E. R. and Everhart, C. F. Design and specification of the Cellular Andrew environment. Technical Report CMU-ITC-070, Information Technology Center, Carnegie Mellon University, August 1988.