Accessing Files in an Internet: The Jade File System

Larry L. Peterson
Herman C Rao[1]

TR 90-30

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF
ARIZONA
TUCSON ARIZONA

Accessing Files in an Internet: The Jade File System

Larry L. Peterson
Herman C Rao[1]

TR 90-30

Abstract

Jade is a new distributed file system that provides a uniform way to name and access files in an internet environment. It makes two important contributions. First, Jade is a logical system that integrates a heterogeneous collection of existing file systems, where heterogeneous we mean that the underlying file systems support different file access protocols. Jade is designed under the restriction that the underlying file systems may not be modified. Second, rather than providing a global name space, Jade permits each user to define a private name space. These private name spaces support two novel features: they allow multiple file systems to be mounted under one directory, and they allow one logical name space to mount other logical name spaces. A prototype of the Jade File System has been implemented on Sun workstations running Unix. It consists of interfaces to the Unix file system, the Sun Network File System, the Andrew File System, and FTP. This paper motivates Jade's design, highlights several aspects of its implementation, and illustrates applications that can take advantage of its features.

January 24, 1991

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

[1]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.

1 Introduction

Workstations connected to national internets have access to a rich collection of resources, including file systems, databases, directory services, supercomputers, and other specialized hardware. An important factor that limits users' ability to take advantage of these resources is the problem of naming and accessing these resources in a transparent and uniform way. This paper describes the Jade file system which addresses this problem in the context of file systems.

Although there are many examples of file systems implemented in distributed environments[Pope85][Howa88][Nels88][Cabr88][Pike90], it is unlikely that a single file system will ever be universally accepted in an internet environment. The main limiting factor in providing such a file system is that the internet spans a collection of autonomous organizations, each of which administers its own system. It is unrealistic to expect all such organizations to adopt a single file system. Thus, the key problem that we must contend with is *autonomy.*

Assuming that the existence of multiple file systems is unavoidable, the problem one must consider is how to integrate these multiple, autonomous, heterogeneous file systems in a way that provides a uniform interface to the files. The Jade file system allows users to construct a private file naming hierarchy out of an existing collection of file systems, and thus access files in a uniform way. We have built a prototype of Jade that integrates Unix file systems, Sun Network File Systems (NFS)[Sun86], and Andrew File Systems (AFS)[Howa88], as well as arbitrary file systems accessed through FTP[USC85].

Jade has two important characteristics. First, it is a logical system that integrates a heterogeneous collection of existing file systems. It dose not provide any storage of its own; it only maps file names onto files that are stored in existing file systems. These underlying file systems may be heterogeneous in the sense that they support different file access protocols; Jade provides the user with a completely homogeneous name space. Furthermore, because of autonomy, 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 instantiation of the Jade File System as a regular file system user without any special privileges. Second, rather than provide a global name space, Jade permits each user to define a private name space. A given user has the same view of heterogeneous, internet-wide file systems, regardless of what machine he or she is using. Jade name spaces support two novel features: they allow multiple file systems to be mounted under one directory, and they allow one logical name space to mount other logical name spaces.

The remainder of this paper is organized as follows. Section 2 motivates Jade's design and contrasts the design with related systems. Section 3 describes the data structures and algorithms used to implement the name space and Section 4 presents several examples that illustrate how one takes advantage of Jade's unique features. Finally, Section 5 discusses Jade's performance and Section 6 offers some conclusions.

2 Client-Based Design

This section motivates Jade's design and compares this design with related systems. The discussion is from both the user's perspective and in terms of the system structure.

2.1 User's Perspective

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. In Jade, users choose the physical file systems they want to access, and glue these systems together to form their own private, logical file systems. A Jade file name, rather than being global, has *scope* relative to a single logical name space. That is, every resolution of file names is performed in the context of a specific user's name space. Figure 1 illustrates an example of a Jade file system associated with the user John, where the dotted lines denote the partitioning of the logical file system into a set of physical file systems.

A collection of physical file systems are *mounted* in a user's private name space. We consider a physical file system to be a service that maps file names to internal handles, and stores or retrieves file data associated with a given internal handle. Each physical file system is addressed by the network address of the host where the physical file system is located and the identifier of the physical file system used by the host. We also assume that each physical file system supports one or more remote access protocols and represents files as non-typed byte-streams. Note that the existence of the network is not totally transparent to the user. The user must know where a physical file system is located to be able to mount it on his or her logical file system. Once a file system is mounted, however, the user can use the logical file system in a network transparent way.

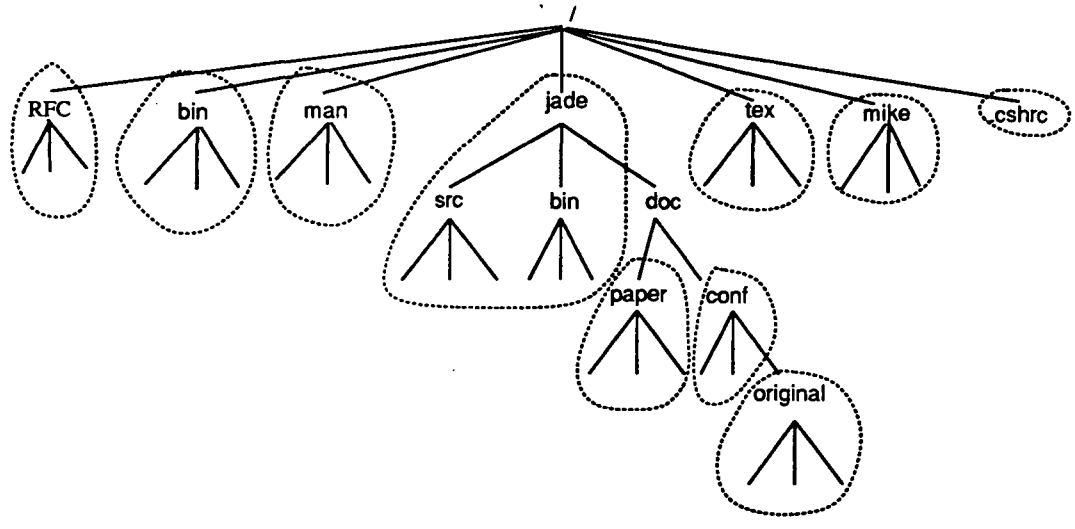

Figure 1: Example Private File Hierarchy

Defining the file system on a per-user basis is well justified. First, it increases user mobility in the sense that users view the same file systems regardless of what workstations they are using, and where

the physical file system is located. Even when users access the network from a single workstation, network window systems (e.g. X Window and Sun NeWS) encourage them to access more than one host at one time. The private file system provides the user with a single name space among these hosts. Second, 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[Floy86][Shel86][Cabr88]. 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.

This per-user name space approach trades the burden on an administrator of maintaining a global name space for the burden on the user of organizing a private name space. Maintaining a consistent and coherent global name space in a large distributed system is not a trivial task. In NFS, for example, if all the hosts mount each other, the number of mount points in the system is proportional to the square of the number of hosts in the environment, producing a significant maintenance overhead. This overhead is probably the limiting factor in the scale of the environment. 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 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 logical file systems by mounting the desired file systems into their logical name spaces.

As mentioned before, the pathname resolution is performed in the context of a specific user's name space. When running a program, by default, the name space of the user who invoked the program (called the *invoker*) is used to resolve names. However, Jade introduces a new feature, called **SetNameSpace**, that allows users to associate a particular name space with a program. This attached name space is used for name resolution when the program is executed. The function **SetNameSpace** is similar to the function **setuid** provided by Unix, which allows changing the privilege of the 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 invokers, to resolve font file names and source file names. The attached name space can also be a special name space defined only for this program. For example, a front-end program of a database system can define its own name space to match internal file organizations.

Like most distributed file systems, Jade supports a *mount* mechanism for gluing together a set of file name spaces into a seamless, hierarchically structured, 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 the problem of autonomy, all of the information necessary to mount one directory under another is maintained in the Jade file system; none of the underlying physical file systems are aware of the fact that they are participating in some user's logical file system. The following subsections introduce the novel aspects of Jade's mount mechanism.

2.1.1 Mounting Logical Name Spaces

To facilitate file sharing, Jade allows one logical file system to be mounted in another Jade file system, in the same way that a physical file system can be mounted into a Jade file system. This allows each user to name files *through* another user's name space. In John's name space shown in Figure 1, for example, the directory **/mike** might refer to a logical name space belonging to another user Mike. Simply by concatenating the leading path **/mike** with the names used by Mike, John can name files using Mike's name space. Moreover, the name space that is mounted by one name space might mount yet another name space. Unlike the Sun Network File System, the Jade file system allows users to name files across name space boundaries. Not only does this feature support file sharing, but it also encourages users to generate *auxiliary* name spaces for special purposes. Section 4 presents one application that takes advantage of auxiliary name spaces.

By mounting logical name spaces, Jade file systems are *linked* to each other to form a loosely coupled confederation. Figure 2 depicts the relationship between one logical name space and a collection of other physical and logical name spaces. 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].

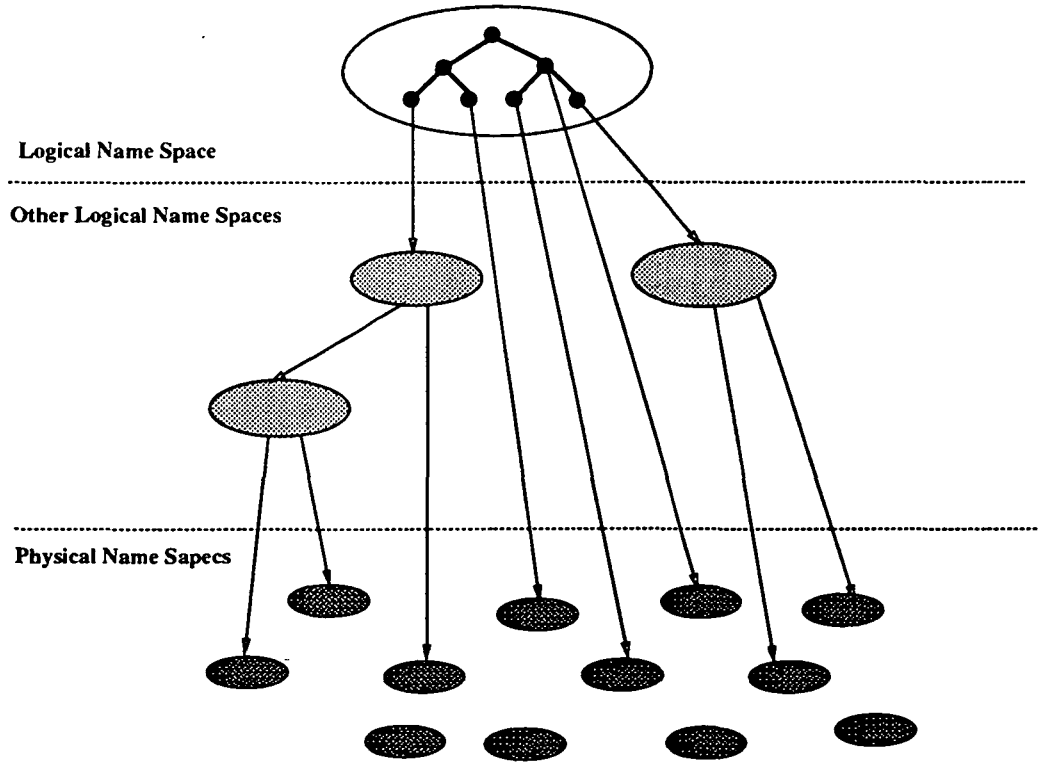

Figure 2: Relationship Among A Collection of Logical and Physical Name Spaces

2.1.2 Multiple Mounts

Jade enhances the functionality of the mount operation by allowing an ordered list of file systems to be mounted under a single directory. In John's name space shown in Figure 1, for example, the directory **/bin** refers to three physical file systems: **/usr/john/bin** in the host **jag**, **/usr/john/bin** in the host **meg**, and the **/usr/bin** in the host **meg**; entries of this directory includes those from these three file systems. This feature, called a *multiple mount*, has a number of advantages, especially when compared with auxiliary mechanisms built on the top of the file system. This is because all directory services (commands) are still applied to directories created by the multiple mount. For example, the multiple mount is capable of supporting the same functions provided by the *search path* or *version file* mechanism; however, by the standard directory listing command (e.g., ls in Unix), users can list all available files under the directory created by the multiple mount. Multiple mounts are especially useful in software development; they can be used to handle version control, software-configuration dependencies, and software distribution[Pete91]. Section 4 describes a version control mechanism built on top of Jade that takes advantage of this feature.

2.2 System Structure

The Jade file system provides a logical layer between file systems and their users. It consists of two major pieces: a Name Space Manager and a Access Manager. The Name Space Manager provides a directory service that maps logical file names provided by the user into handles for the files; it is called when opening files. The Access Manager supports file access, given a file handle.

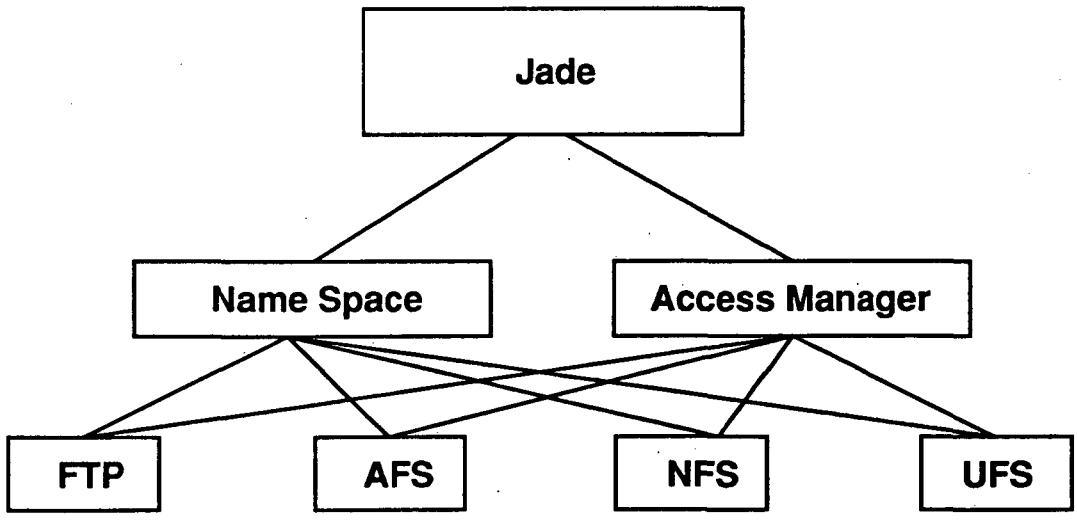

Figure 3: Relationship between Jade and Other File Access Protocols

In Jade, 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,

depend on the underlying file access protocols. For example, a Jade file system might depend on NFS, AFS, FTP, and the Unix file system (UFS for short) as illustrated in Figure 3. The Name Space Manager uses the directory capabilities of those protocols (e.g., their ReadDir, Lookup, and GetAttr operations), while the Access Manager invokes their storage-related operations (e.g., their Read, Write, Get and Put operations). The reason behind this separation is based on the observation that the *directory* abstraction is different in dissimilar file systems. In fact, most file system protocols provide a completely different set of operations for directory services.

2.2.1 Access Manager

Jade's Access Manager is similar to the Cache Manager used by the Andrew File System. When a file is opened, the Access Manager checks the cache for the presence of a valid copy. If such a copy exists, the cached copy is opened and used. Otherwise, an up-to-date copy is *fetched* from the 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 *restored* back to the physical file system when the file is closed.

Jade differs from the Cache Manager of the Andrew File System in three 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. 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 advantage of this refinement is that it allows use of the Jade file system even without a local disk for caching. Second, Jade does not commit to one single protocol suite as does Andrew. Instead, the Access Manager behaves as a switch that selects from among several access protocols. As illustrated in Figure 3, Jade accommodates access protocols of NFS, AFS, FTP, and UFS. Third, Jade separates the fetch (restore) operation from the open (close) operation. The fetch and restore operations are defined by the access protocol of the underlying file system where the file is located. After the file is fetched, 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.

As shown in Figure 4, the file object maintained on the logical file system includes two caching operations (fetch and restore), several access operations (open, read, write, seek, close, and so on), and two references (to the cached copy and to the original file). Since the design and implementation of the uniform interface is straightforward, the rest of this paper concentrates on issues of naming.

2.2.2 Name Space Manager

The Name Space Manager implements a *skeleton hierarchy*. This hierarchy keeps track of the boundaries between the underlying file systems. It is only a skeleton because most of the directory nodes within a given subtree are maintained by some physical file system, not by the Name Space Manager. Only a few directory nodes, called *skeleton nodes*, are maintained by Jade. Another way of saying this is that the skeleton hierarchy is superimposed over a collection of existing file hierarchies; much of the structure of the underlying hierarchies remain visible to the user.

Each skeleton node maintains a list of references to other file systems. The mount operation is

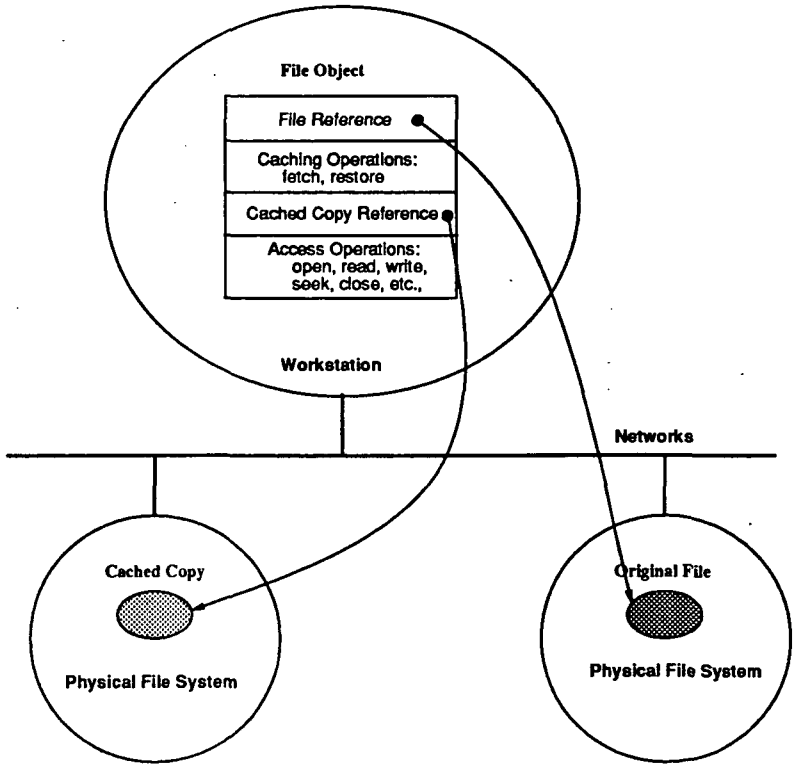

Figure 4: File Object

used to attach given file systems to the name space by creating a skeleton node. Jade generalizes the mount operation to allow none, one, or more than one file system to be attached to a single skeleton node. Moreover, the mounted file system pointed to by the reference can be either a physical file system or another Jade file system. When resolving a given pathname, Jade first locates the proper skeleton node, and then resolves the rest of the pathname by consulted the underlying file systems referred to by references stored in the skeleton node. However, since a given skeleton node 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. Section 3 discusses pathname resolution in more detail.

Although conceptually simple, this design is more powerful than techniques introduced by other distributed file systems; e.g., *prefix tables* and *remote links* used by the Sprite File System[Welc86], *mounting tables* and *mount points* used by NFS, 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 which 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 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. Sun's 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 workstation, while the latter is maintained by individual physical file systems. Jade, on the other hand, realizes the mounting relation as the skeleton node and maintains it at only the client workstation. 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 node under another is maintained at the client.

In most other distributed file systems, the name space is implemented as a kernel service. In Jade, on the other hand, the private name space is implemented as a separate name server; each user process uses an interprocess communication mechanism to consult its private name space. Since the name space is defined on per-user basis, the traffic to an individual name space is small and limited, and the private name space is not the bottleneck of the system. The advantage of this approach is that it allows a set of processes owned by one user to share the same naming environment even when those processes span more than one host.

The skeleton node can be considered a generalization of *symbolic links* provided by most Unix-like File Systems[Ritc78][Saty85][Welc86]. A symbolic link is applied only within the file system in which it resides; it redirects a path from one subtree to one other subtree in the same name space. Symbolic links are, therefore, always leaves in the naming tree. On the other hand, a skeleton node refers a path to another file system. A skeleton node may even point to more than one file system. Moreover, one skeleton node can be under another skeleton node. For example, the skeleton node **/jade/doc/conf/original** in Figure 1 is under another skeleton node **/jade/doc/conf.**

Projects like Tilde[Come85], QuickSilver[Cabr88], and Plan 9[Pike90] provide mechanisms to let users construct their own name spaces rather than a single global name space. However, Jade surpasses these systems in their ability to accommodate heterogeneity, allow for customization, and support interactions between name spaces. Generally, none of these three systems allow a name space to be mounted into another name space, and they all commit to a single protocol-suite. Tilde allows users to choose individual name spaces (called *trees*) to form their naming environment (called the *forest*). However, it does not allow one tree be attached under another tree, and therefore the pathname is always started from the tree's name. Plan 9 provides a per-process based name space. However, whenever invoking a new job in other servers, it needs to reconstruct a new naming environment, thereby trading the cost of querying a separate name server for the cost of generating a new naming environment.

3 Name Space

This section describes the data structures and algorithms used to implement the name space. The key data structure is the *skeleton hierarchy*, which maintains boundaries between the underlying file systems. A *mount* operation is used to create and tailor this hierarchy. The pathname algorithm is more complex than those used by other file systems. This is because a given logical directory may point to multiple underlying file systems, as well as to other logical file systems.

3.1 Skeleton Hierarchy

A Jade name space consists of a collection of subtrees, or *domains*. Each domain corresponds to one underlying file system. Domains are glued together to form the Jade name space. The root of a domain—the node by which the domain is attached to the name space—is called a *skeleton node*. For example, the domain rooted at **/jade** represents one domain and the node **/jade** is a skeleton node in Figure 1.

A given skeleton node contains three pieces of information: its pathname, its attributes, and a list of references to directories in other name spaces. The reference identifies the host (or the *cell* for AFS) that implements the desired file system, the root in the mounted file system, and the network access protocol that is used to access this file system. For example, the skeleton node with the pathname **/jade** (called the skeleton node **/jade**) is given by

</jade, 0666, meg.cs.arizona.edu:/usr/john/jade:NFS>

In this example, **meg.cs.arizona.edu** (or **meg** for short) is the host name; **/usr/john/jade** is the root of the mounted file system; and **NFS** indicates the NFS protocol is used to access files on **meg**. As another example, the skeleton node **/jade/doc/paper** is given by

</jade/doc/paper, 0666, cs.arizona.edu:user.john:/afs/az/usr/john/paper:AFS>

where **cs.arizona.edu** is the cell name; **user.john** is the volume name of the mounted file system and **/afs/az/usr/john/paper** is the name of the root; and **AFS** indicates the AFS protocol used to access files in this domain. The skeleton node might refer to another Jade file system rather than a physical file system. For example, the skeleton node **/mike** refers to another Jade file system named **mike@state.edu** and is given by

</mike, 0666, mike@state.edu:/database:JNP>

where **JNP** (Jade Naming Protocol[Rao91]) is used to access Jade name space **mike@state.edu**. Furthermore, the skeleton node might have zero, or more than one, reference. For example, there are three references associated with the skeleton node **/bin**:

jag.cs.arizona.edu:/usr/john/bin:UFS
meg.cs.arizona.edu:/usr/john/bin:NFS
meg.cs.arizona.edu:/usr/bin:NFS

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

meg.cs.ariozna.edu:/usr/john/RFC:NFS
nic.ddn.mil:RFC:FTP

On the other hand, the skeleton node **/jade/doc** has no references and is considered as a *logical* directory with two entries: **/jade/doc/paper** and **/jade/doc/conf**. The root directory (/) is another example of a skeleton node without any references. Notice that skeleton nodes with no references are not necessarily empty directories in the logical hierarchy; they may contain other skeleton nodes.

The *mount* operation is the basic operation provided by Jade to allow users to construct their name spaces. It creates a skeleton node in the user's logical hierarchy with the given path name, and associates this node with references to file systems. In addition to the traditional mount that links a node to one single file system, Jade supports ***null mount*** and ***multiple mount*** options. With the null mount option, no file system is bound to the skeleton node. With the multiple mount option, the node is bound to an ordered list of references. The order is important to resolve name conflicts. We will discuss the name conflict issue later. Also, when new files are created, they are stored on the first physical file system indicated in the list.

One Jade name space can be mounted into another Jade name space in the same way that a physical file system can be mounted into a Jade name space. For example, skeleton node **/mike** in John's name space as shown Figure 1 mounts the subtree **/database** of a private name space belonging to another user named Mike. The Jade file system provides JNP for accessing the logical name space. 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. The Jade file system also allows users to name files across name spaces. As illustrated in Figure 5, Mike's name space mounts the subtree **/bib/journal** of Bob's name space under the skeleton node **/database/ieee**. Therefore, 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.

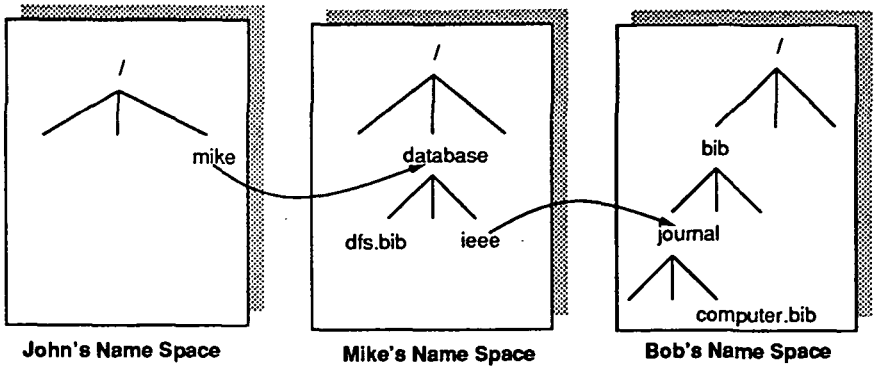

Figure 5: Example Mounting Other Name Spaces

Figure 6 lists all directory operations provided by the name space. Notice that unlike most Unix-like file systems, which treat directories the same as files, Jade directories are treated differently from regular files. Specifically, file access operations (i.e., open, read, write, close, and seek) are no longer applied to directories. This is necessary because directory services are supported by Jade, rather than by the underlying physical file systems.

Using Figure 1 as an example of a private name space, consider the following examples. First, the existence of skeleton nodes is not hidden from users. For example, the user can list entries in a directory as follows:

```
% ls -l /jade
```

mkdir	Makes a new directory in the physical file system.
rmdir	Removes a directory in the physical file system.
mount	Creates a skeleton node in the logical file system.
unmount	Removes a skeleton node in the logical file system.
lookup	Returns the file handle for the named file.
remove	Deletes the specified file.
getentry	Lists the entries under one directory.
getskeleton	Lists the skeleton nodes under one directory.
getattr	Gets the attributes of a file/directory.
setattr	Sets the attributes of a file/directory.

Figure 6: Directory Operations

```
Sdrwxrwxrwx   john   ./        →<meg:/usr/john/jade:NFS>
-drwxrwxrwx   john   src       →<meg:/usr/john/jade/src:NFS>
-drwxrwxrwx   john   util      →<meg:/usr/john/jade/util:NFS>
Sdrwxrwxrwx   john   doc       →<>
```

Note that the mode of files and directories has been modified to include skeleton information. The first character of the mode string S indicates that the entry is a skeleton node. The last column of a skeleton node entry specifies references to a set of other name spaces.

Second, it is important to distinguish between the operations that modify the underlying file systems and those operations that modify the logical hierarchy. For example, the operation **mkdir** creates a new directory in a physical file system, as illustrated in the following example.

```
% mkdir /jade/tmp
% ls -l /jade
Sdrwxrwxrwx   john   ./        →<meg:/usr/john/jade:NFS>
-drwxrwxrwx   john   src       →<meg:/usr/john/jade/src:NFS>
-drwxrwxrwx   john   util      →<meg:/usr/john/jade/util:NFS>
-drwxrwxrwx   john   tmp       →<meg:/usr/john/jade/tmp:NFS>
Sdrwxrwxrwx   john   doc       →<>
```

That is, a new directory named **tmp** is created in the physical file system **/usr/john/Jade** on host **meg**. In contrast, the operation **mount** creates a skeleton node that exists only in the skeleton hierarchy.

```
% mount /jade/menu jag:/usr/john/menu
% ls -l /jade
Sdrwxrwxrwx   john   ./        →<meg:/usr/john/jade:NFS>
-drwxrwxrwx   john   src       →<meg:/usr/john/jade/src:NFS>
-drwxrwxrwx   john   util      →<meg:/usr/john/jade/util:NFS>
```

```
-drwxrwxrwx   john   tmp    →<meg:/usr/john/jade/tmp:NFS>
Sdrwxrwxrwx   john   doc    →<>
Sdrwxrwxrwx   john   menu   →<jag:/usr/john/menu:UFS>
```

The first argument to **mount** gives the logical directory name. The second argument identifies the file system that is to be mounted at the newly created directory. If the second argument is not given (i.e., the mount operation with the null option), then the directory exists only in the logical hierarchy, as is the case with **/jade/doc**. If there are more than two arguments (i.e., the mount operation with the multiple option), the command creates a directory mounting more than one file system. Notice that the protocol used to access the file system is determined by Jade; it is not specified by the user. Jade maintains a host/protocol table, and uses this table to complete the mounting information.

Entries of the directory created by the multiple mount are composed of the union of all specified mounted file systems. Consider the following mount operations:

```
% mount /jade/bin.meg meg:/usr/john/bin
% mount /jade/bin.jag jag:/usr/john/bin
% mount /jade/bin meg:/usr/john/bin jag:/usr/john/bin
```

The files under the directory **/jade/bin** consists of the directories **/jade/bin.meg** and **/jade/bin.jag** as follows:

```
% ls /jade/bin.meg
efind       find       jdacc
% ls /jade/bin.jag
iostat      jdacc      pnsd
% ls /jade/bin
efind       find       iostat
jdacc       pnsd
```

Notice that the filename **jdacc** exists in both of the file systems **meg:/usr/john/bin/jdacc** and **jag:/usr/john/bin/jdacc**, and **/jade/bin/jdacc** refers the one located on the former file system. Also, new files are created on the former file system as shown in the following example:

```
% touch /jade/bin/bar
% ls /jade/bin
bar         efind      find
iostat      jdacc      pnsd
% ls /jade/bin.meg
bar         efind      find       jdacc
% ls /jade/bin.jag
iostat      jdacc      pnsd
```

Jade supports two operations to list entries of a directory: **getentry** returns all entries (skeleton nodes and nodes on physical file systems) under the directory, and **getskeleton** returns only local skeleton nodes under the directory. Since a given directory may refer to more than one physical file system, the cost of collecting all entries can be very high. Jade provides **getskeleton** as a less expensive alternative, and as Section 3.2 points out, this operation is very useful in pathname resolutions.

3.2 Pathname Resolution

In order to resolve a given pathname, Jade locates the desired domain by identifying the skeleton node whose pathname has the longest matched prefix with the given pathname. Jade then resolve the rest of the pathname by consulting the underlying file systems referred to by the list of references associated with this skeleton node. If there is only one physical file system specified by the list, the procedure to resolve the remaining path is straightforward. However, because of multiple mounts and mounting logical name spaces, this procedure is more complicated. Three issues need to be considered. The first issue is the simple case of resolving the pathname in 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 deals with the multiple mount. For the multiple mount, it may be necessary to try several possibilities before successfully resolving the given name.

3.2.1 Resolving Pathname in Physical File Systems

In general, there are two approaches in resolving the remaining path in the physical file system. In the first approach, called *local pathname resolution*, each directory is brought across the network from the host of the physical file system and searched on the host of the logical file system. In the second approach, called *remote pathname resolution*, the pathname is packaged into a network request message and sent to the host of the physical file system, which then opens and searches directories locally.

The Jade file system supports the local pathname resolution model. In order to improve performance, Jade not only maintains the skeleton hierarchy but it also caches the remote directories. The reason behind this decision is that experiments[Shel86][Howa88] have shown that the activity of most users is confined to a small slowly changing subset of the entire name space hierarchy. Thus, a directory cache on the Jade has a high hit ratio, and much network traffic for moving directory entries from remote file systems is avoided.

Caching is a general technique for reducing the cost of pathname resolution in distributed systems[Shel86][Terr87][Saty90]. However, its functionality varies in different file systems. For example, prefix tables used by the Sprite File System only map pathname prefixes to file servers; NFS caches attributes and file handles of visited files/directories for later access; Locus and Andrew use the local pathnames resolution and cache intermediate directories when resolving pathnames. Like Locus and Andrew, Jade caches intermediate directories. However, rather than starting from the root and caching each component directory under the root like Locus and Andrew, Jade starts from the skeleton node and caches only component directories under this domain.

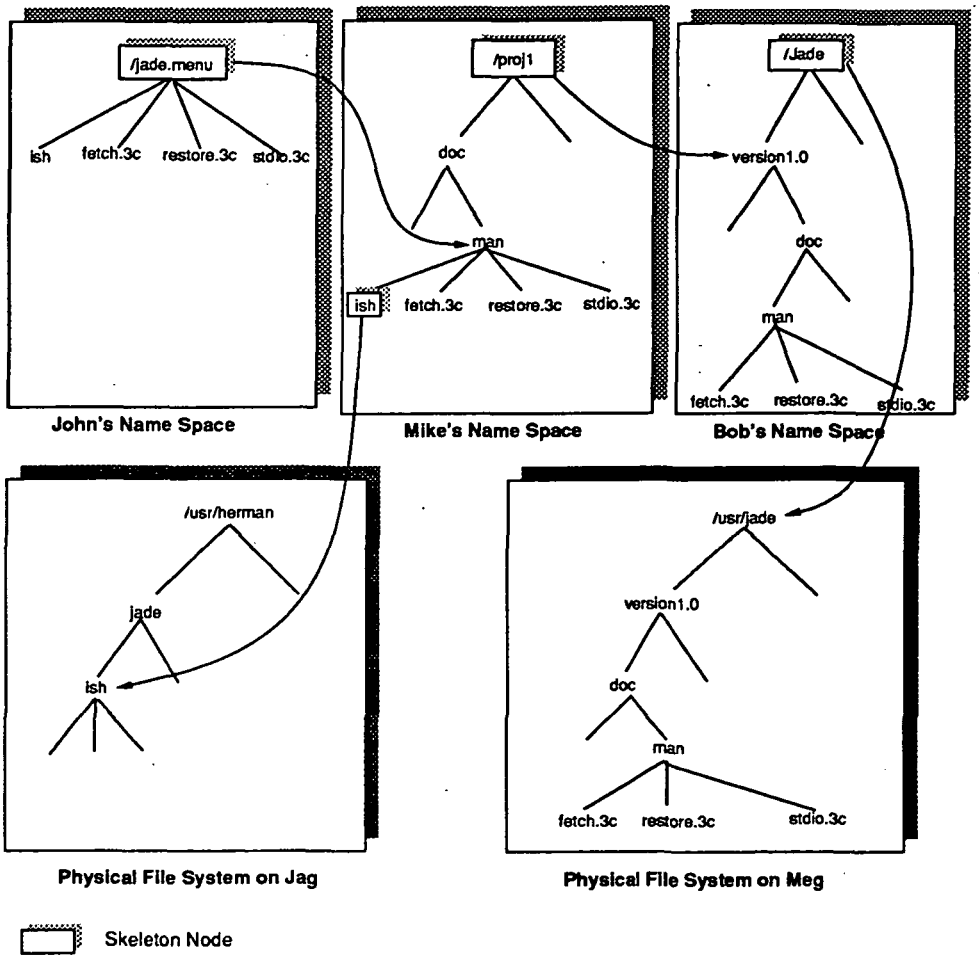

Figure 7: Example Multiple Logical Name Spaces

3.2.2 Resolving Pathname in a Sequence of Name Spaces

Mounting logical file systems can be treated the same as mounting physical file systems, but the resolution procedure is more complicated when a sequence of logical name spaces may need to be traversal before a desired physical file systems can be located. Consider Figure 7 which shows three logical name spaces (i.e., John, Mike, and Bob) and two physical file systems (i.e., Jag and Meg), where John's name space mounts Mike's name space, which in turn mounts Bob's name space, which finally mounts a physical file system on **meg**. Each of the directories **John:/jade.menu** (the directory **/jade.menu** on John's name space), **Mike:/proj1/doc/man**, and **Bob:/jade/version1.0/doc/man** refer to the same physical directory: **meg:/usr/jade/version1.0/doc/man**. But in order to resolve a pathname **/jade.menu** from John's name space, it is necessary to consult each of the logical name spaces before the physical file system is found. Notice that nodes in the name spaces of this sequence may contain other skeleton nodes. For example, Mike's name space has a skele-

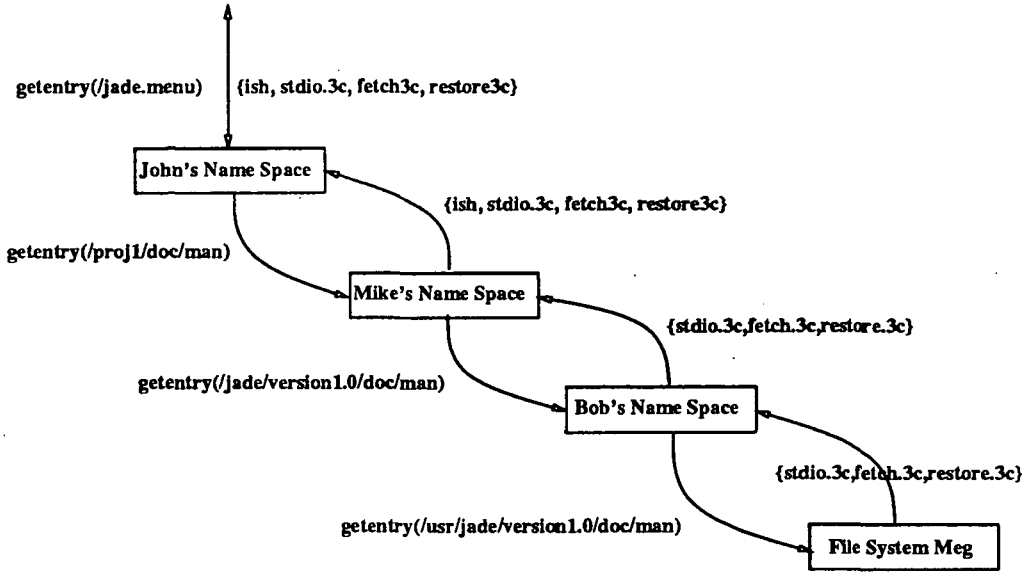

Figure 8: Example Recursive Method

ton node **/proj1/doc/man/ish** introducing a different domain, and therefore, the directory **Mike:/proj1/doc/man** (and hence **John:/jade.menu**) includes not only files in the directory **meg:/usr/jade/version1.0/doc/man** (i.e., **fetch.3c**, **restore.3c**, and **stdio.3c**), but also **ish**. Also notice that it is possible to form a loop within this calling sequence. How to detect and prevent loops is one critical issues in designing the pathname resolution algorithm. Finally, name conflicts may exist between names of local skeleton nodes and those from mounted name spaces. We will address these two issues in later sections.

There are two methods to resolve the pathname in a sequence of name spaces: the *recursive* method and the *iterative* method. With the first method, pathnames are recursively resolved within the new name space, and all the entries (including local skeleton nodes and mounted files) of the directory are collected and returned to the caller one at a time. Figure 8 illustrates the procedure to query the directory **/jade.menu** of the name space **John** using this method. The query starts from John's name space, which then generates a new query to Mike's name space, which in turn queries Bob's Name space, which finally consults the physical file system **meg**. The answers come backward from **meg** to **Bob** to **Mike**, and finally to **John**. The recursive method has the advantage of the forward mounting property being completely hidden from the current name space: the *getentry* operation is the only directory lookup service provided by the logical name space, and the procedure for handling logical file system mounting is treated in exactly the same way as that of the physical file system mounting. However, this method is very expensive since it requires each of the logical name spaces in the calling sequence to collect directory entries before returning the query. Moreover, because of its recursive nature, the original name space has no control over the whole resolution activity, making detection of loops in the mounting sequence more difficult.

We chose the iterative method illustrated in Figure 9. With the iterative method, the original logical name space (i.e., John's name space) retains control over the resolution activity. When

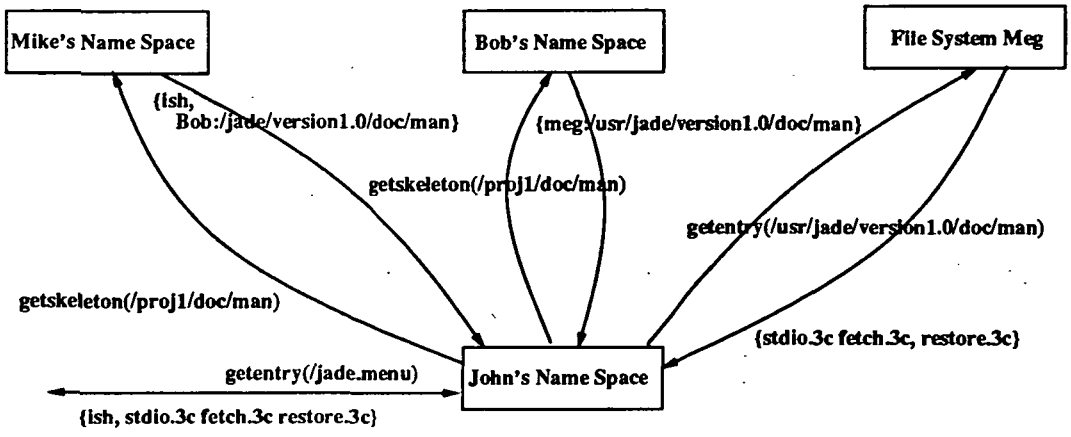

Figure 9: Example Iterative Method

Jade calls a given name space, that name space responds to the query with the list of references associated with the queried node and local skeleton nodes under this node. As in the example, Mike's name space returns the query of the pathname **/proj1/doc/man** with the reference to **Bob:/jade/version1.0/doc/man** and the skeleton node under the node **/proj1/doc/man**. John's name space then queries Bob's name space for further information, and so on. In this method, the skeleton nodes are exposed, rather than hidden, by each logical name space. The *getskeleton* operation queries the skeleton nodes under one directory (called the *skeleton children* of the directory) and the list of references associated with this directory. In contrast, the *getentry* operation lists all entries under one directory, including skeleton nodes and regular files and directories. Since the original name space has full control over the resolution procedure, it is easy to detect loops in the mounting graph.

3.2.3 Handling Multiple Mounts

Jade allows more than one file system to be mounted on one single skeleton node. 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 *direct graph* that describes the mounting relationships. Figure 10 illustrates a node of the name space **A** and all name spaces referenced by this node. In this example, name spaces are considered as units: the name spaces of **A**, **C**, and **F** are logical name spaces, while the name spaces of **B**, **D**, **E**, **G**, **H**, and **I** are physical file systems. Arrows represent mounting relationship among name spaces. In this example, a skeleton node in the name space **A** multiply mounts nodes in name spaces **B**, **C**, and **D**.

The mounted file system (either physical file system or a logical file system) may have files with names that already existed in the original skeleton hierarchy. Also, files from different file systems mounted on one skeleton node (multiple mount) may have the same name. Jade uses the following two rules to resolve name conflicts. First, names of local skeleton nodes have precedence over names from mounted files systems. Second, when there are conflicts from different mounted file systems,

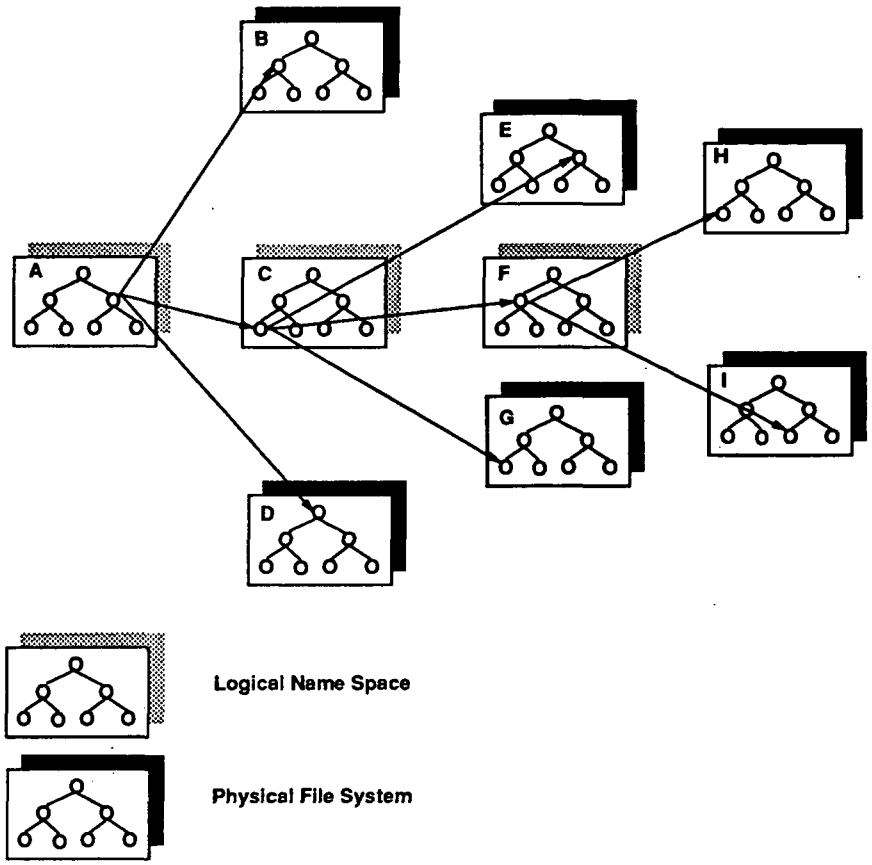

Figure 10: Example Direct Graph

the order of the list of references associated with the node is used to resolve conflicts[1]. Thus, names from the mounted file system appeared in the front of the list of references have preference over those appeared in the back of the list.

These preference rules suggest that the 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**. In order to improve the performance, the searching procedure halts whenever the desired name is found in one of name spaces of this sequence.

3.2.4 Pathname Resolution Algorithms

To summarize, the Jade file system maintains the skeleton hierarchy and caches entries of visited remote directories; a given skeleton node points to zero or more file systems, each file system may be either a physical file system or another Jade file system; Jade uses local pathname resolution, caches remote directories, iteratively consults individual name spaces, and uses depth first search to consult name spaces. This subsection completes the picture by presenting the algorithm used to

[1] An alternative way to resolve name conflicts between different mounted file systems is to compare timestamps of files with the same name. It therefore changes the semantics of the list of references.

resolve a pathname.

A Jade name space is implemented by a collection of nodes, each of them is either a skeleton node or a *cached node.* Cached nodes, in turn, correspond to either skeleton nodes in other Jade name spaces, or files/directories in physical file systems. The structure of the node consists of the pathname, a list of references to other file systems, and a set of attributes. A hash table is used to locate nodes by mapping a given pathname into the corresponding node.

The pathname resolution function, ResolvePathName()[2], translates a pathname to the pointer of the node. ResolvePathName() is given in the Appendix. It starts by locating the node in the current name space which has the longest prefix of the input pathname; this node is called the *closest ancestor node.* The *closest ancestor node* can be either a skeleton node or a cached node. FindClosestAncestor() performing this search is given in the Appendix. If the *closest ancestor node* is a cached node, the validity of this cached node is checked. In order to reduce the traffic between Jade and the physical file systems, we examine only the *closest ancestor node* instead of all the nodes along the path from the skeleton node to the *closest ancestor node.*

Once the *closest ancestor node* is located, ResolvePathName() then resolves each component of the remaining pathname. The function maintains two lists: a *outstanding list* keeps outstanding references, and a *visited list* records references that have been visited. The *visited list* is used to avoid visiting previous references in order to prevent the loops in the mounting sequence. The body of ResolvePathName() consists of two while loops. The outer while loop scans each component in the remaining path, and the inner while loop consults each of the references in the *outstanding list* in order to resolve the current component. At the beginning, the *outstanding list* is set to the references associated with the *closest ancestor node.* Whenever the component is resolved, the *outstanding list* is reset to the list of references associated with the new node. For each reference in the *outstanding list,* the name space pointed to by this reference is consulted using the Cache() function. If the reference points to a physical file system, Cache() calls the operation **getentry** to get entries in the remote directory and caches them. If the reference refers to a logical name space, Cache() calls the operation **getskeleton** to get the skeleton children under the remote node and the reference list associated with it, caches these skeleton children, and returns the reference list. This list then is put in front of *outstanding list* in order to implement the depth first search. The inner while exists whenever the node with the name of the current component is located, otherwise the while loop continues. If the *outstanding list* is empty, the function ResolvePathName() fails.

3.3 Other Issues

3.3.1 Get Directory Entries

Unlike other distributed file systems, Jade uses the hash table to locate desired nodes directly as mentioned in the previous section, and it does not maintains any reference from a directory to its entries. This is because entries under a directory include not only nodes from mounted file systems, but also skeleton nodes in the logical name space. Thus, listing entries under a directory in Jade is more complicated than in other systems. The Appendix outlines the *Dir()* function that implements this operation; the complete algorithm is presented in [Rao91]. In addition to the previously mentioned hash table that maps a pathname into a node, another hash table is needed in order to implement this function. This hash table, called the *Skeleton Children Table* (or **SCT**),

[2]ResolvePathName() is logically equivalent to namei() in the Unix operating system.

maps a given pathname into a set of nodes, each of which is an entry under this pathname and is a skeleton node. Using Figure 1 as an example, SCT maps the pathname **/jade/doc** to two skeleton nodes: **/jade/doc/paper** and **/jade/doc/conf**. *Dir()* starts by calling *ResolvePathName()* to locate the directory node corresponding to the input pathname. Then, SCT is used to collect entries which are children under this directory node and are skeleton nodes. Finally, the depth first method presented in the previous section is used to collect other entries from mounted file system referred to by this directory node.

3.3.2 Access Control

The Jade file system 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 file system and user; it does not have any special privileges. We also believe that the authentication mechanism should be installed on the server where objects are implemented, rather than on the intermediate agent. Acting as an agent, the Jade file system also collects all authentication information needed to access file systems and issues the proper information automatically whenever it accesses these files.

One problem of this decision is that a skeleton node 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 that physical file system.

4 Applications

This section presents three examples to illustrate the novel features provided by the Jade File System. The first example shows that Jade provides a rich set of functions that allow users to tailor their private name spaces to fit their needs. The second example illustrates how the mount mechanism can be used to build an architecture specific name space in a heterogeneous environment. The final example describes a version control mechanism built on top of Jade. This mechanism provides a hierarchical view of a collections of files for each programmer and allows a maximal sharing of theses files.

4.1 Tailoring Name Spaces

Since the structures of the underlying hierarchies of file systems remain visible to users, Jade provides methods to allow users to assemble their own name spaces from these various underlying hierarchies, thereby customizing the systems according to their own preferences. There are several ways that users can tailor their name spaces.

First, skeleton nodes might not be part of any physical file system and may serve only as logical directories with entries of other skeleton nodes. These skeleton nodes are created by the mount operation with the null option. For example, the directory **/jade/doc** in Figure 1 is not contained in any physical file system. Also, the resolution procedure implies that the name of a skeleton node has preference over the name of files/directories in physical file systems, implying that names in the private hierarchy *supersede* names in the underlying file systems. For example, if there were a directory named **/usr/john/jade/doc** on host **meg**, then this directory would not be visible

to the user because it would be hidden by the private directory **/jade/doc**. That is, **/jade/doc** *replaces* **meg:/usr/john/jade/doc**. As another example, had there been a file or directory named **/usr/john/jade/junk** on host **meg**, then it would have been hidden by the definition of **junk** in the private directory. Because **junk** is bound to a null physical file system and does not point to another private directory, its only purpose is to *hide* something in the underlying file system. Entries that hide files or directories in the underlying file system are not automatically displayed unless explicitly referenced.

Second, with the multiple mount option, users can define a directory to include files located in more than one file system. The order in which the file systems are mounted is significant. For example, the directory **/bin** points to **meg:/usr/john/bin**, **meg:/usr/bin**, and **jag:/usr/john/bin**. As another example, the directory **/tex** points to **jag:/usr/john/tex**, **jag:/usr/mike/tex**, and **meg:/usr/lib/tex/macros**. This features provides the same functionality as search paths in Unix. The advantage of our approach is that directories created by the multiple mount option are treated exactly the same as other directories, and all directory operations still apply to these directories. For example, with the command "**ls /tex**", the user can list all files under **jag:/usr/john/tex**, **jag:/usr/mike/tex**, and **meg:/usr/lib/tex/macros**, while the command "**ls -l /tex/plain.fmt**" can be used to find out on which physical directory the file **plain.fmt** is located. In contrast, Unix does not provide any general mechanism to list all available files under the search path, or to locate a desired file by its name[3].

Finally, the multiple mount option can be used to locate a file that is replicated in several file systems. If there is a failure that causes one physical file system to become unreachable during pathname resolution, Jade consults the next physical file system in the reference list. For example, the directory **/man** points to **jag:/usr/share/man** and **meg:/usr/share/man**. With this scheme, the physical file systems under the directory are usually identical and read-only. Putting them under the same name, the replication property is transparent to users.

4.2 Architecture Specific Name Spaces

Jade allows a logical name space to be mounted into another name space. Using this scheme, users can create auxiliary name spaces for special purposes. This technique can be used to hide hardware heterogeneity.

Consider a user that uses either a MIPS or a Sun SPARC workstations, both running Unix. The user might have a primary name space **main** which includes binaries for both architectures. However, the user can also create architecture-dependent name spaces **sparc** and **mips**, each of which consists of skeleton nodes pointing to the proper nodes in the name space **main**, as illustrated in Figure 11. In the name space **main**, the directory **/bin/mips** includes binaries for the MIPS architecture, while the directory **/bin/sparc** consists of binaries for the SPARC architecture. The auxiliary name space **sparc** (**mips**) has two skeleton nodes: root (/) pointing to the root of the name space **main**, and **/bin** pointing to **/bin/sparc** (**/bin/mips**) of name space **main**. When the user logs onto the workstation, the appropriate name space (either **mips** or **sparc**) is initiated automatically[4], and the user can use the same name to address the binary regardless of which of the two workstations he or she is using.

[3] Unix provides the command which to locate a given command, but it can only be applied to commands.

[4] In Unix, a simple routine in .cshrc file can perform the initialization.

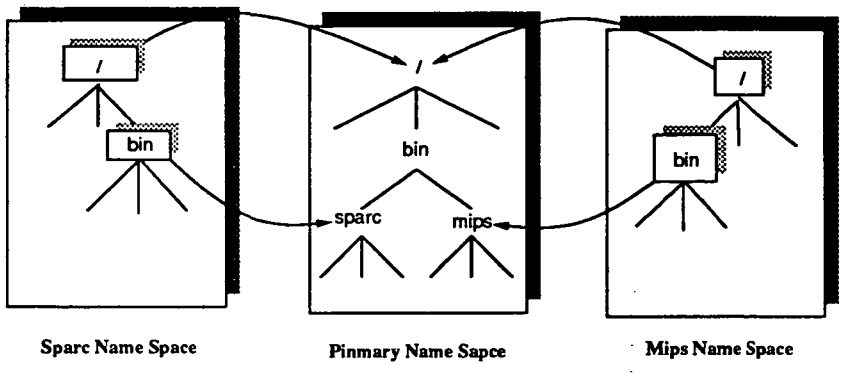

Figure 11: Example Architecture Dependent Name Spaces

4.3 Version Control

This section discusses how a version control mechanism can be easily built on top of the Jade file system; the complete description of this mechanism can be found in [Rao91]. Jade allows users to build their own views of a set of files. The refinement of the mount operation encourages users to reorganize the structures of files in the logical layer rather than the physical layer. It thus provides a framework for a software development and maintenance environment that allows several programmers to work on a set of source files simultaneously.

In a large software project, there may exist more than one version of the software (e.g., one or more release versions, a testing version, a working version for each programmer). In addition to multiple versions, there may be multiple programmers working simultaneously. There are two contradictory tasks in designing a software development environment. First, the system should let users share files in different versions and switch between versions easily. Second, the system should provide a mechanism to let the user build a private working area, without worrying about interference from other programmers. Traditional software like SCCS[Allm86] require that a complete copy of all the source files be made every time a new working area is needed. It can be very expensive to copy files, especially for a large set of source files.

Recall that Jade pathnames are resolved relative to private name spaces and the multiple references associated with one skeleton node provide a hierarchical view of a set of files located in different physical directories (even on different hosts). For example, Figure 12 shows a software development environment in which two programmers, John and Mike, share files located in four different versions, but each programmer has his own view of the files. Consider, for example, John's view consisting of three references. The first points to the directory **jag:/jade/working/john/src**, the second refers to directory **zep:/jade/test_version/src**, and the last points to the directory **meg:/jade/release_version/src**. John's view of the directory **/project/src** is shown in Figure 13. The system also provides functions to let users adjust their view, as well as get more information about this view. For example, John can change his current view from the one he is working on to the test version simply by removing the first reference in the order list.

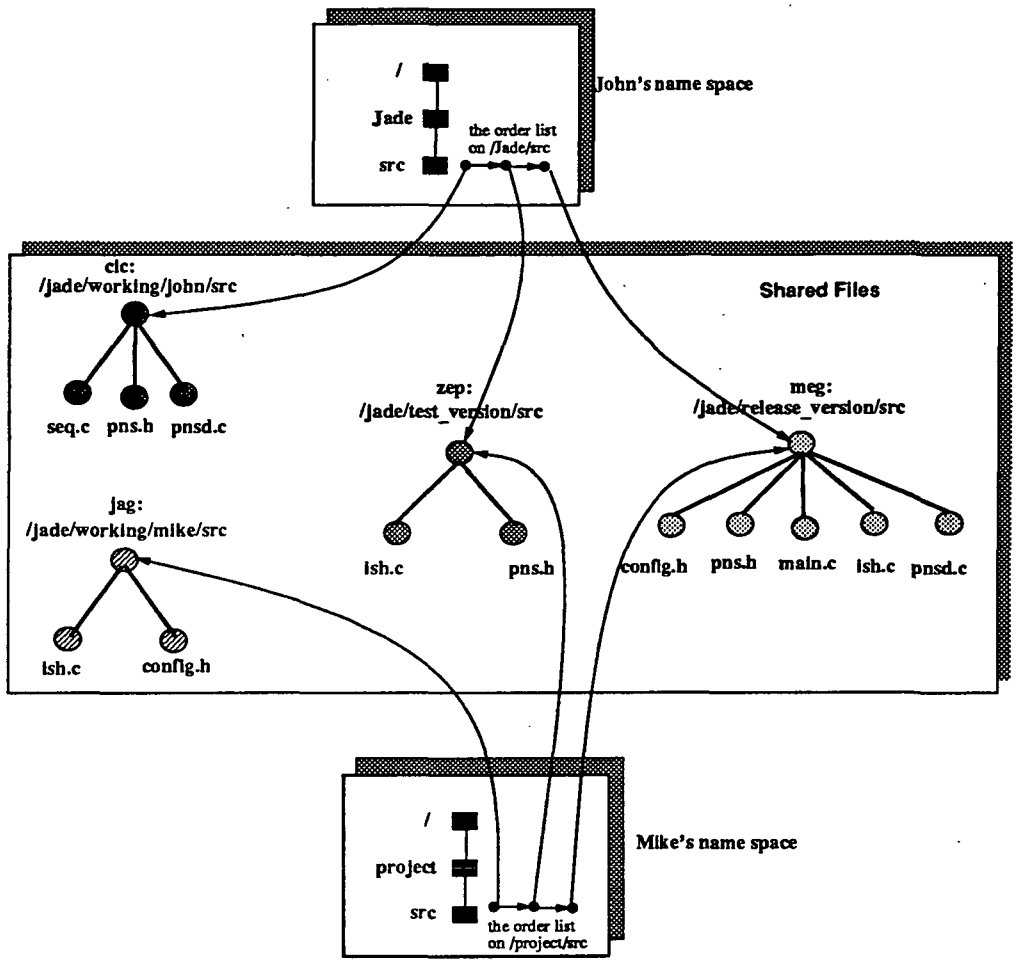

Figure 12: Example Software Development Environment

5 Performance

We have implemented a prototype of the Jade File System on the Sun workstation running Sun OS 4.1. The prototype consists of interfaces to UFS, NFS, AFS, and FTP. It is implemented in such way that it co-exists with the file system provided by the kernel. We measured the performance of this prototype using the Andrew Benchmark[Howa88]. This section briefly describes the prototype and then discusses its performance.

5.1 Prototype

Jade is implemented at the user-level without any modification to the kernel. It uses Sun RPC as the interprocess communication mechanism between the components illustrated in Figure 14. The Name Space Manager and the Access Manager are implemented as separate severs; the former resolves the pathname, while the latter handles file caching. Although the Name Space Manager is defined on a per-user basis, the Access Manager is designed in such a way that processes from

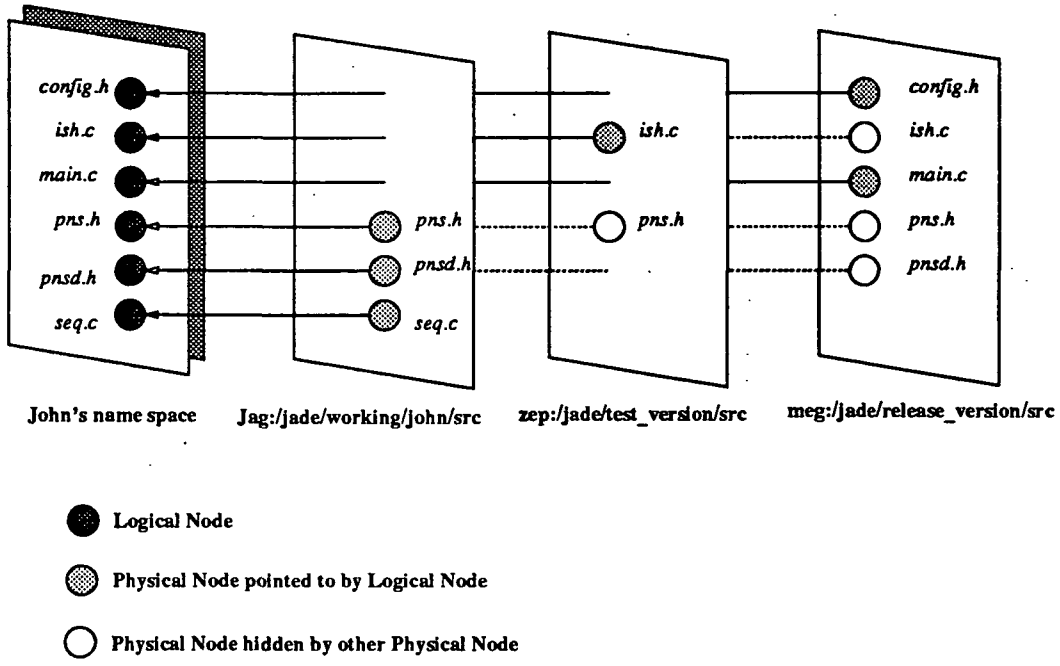

Figure 13: Example of Hierarchy View

different users can share cached files, even if they are located on distinct hosts. The C library is modified to include calls the Name Space Manager and the Access Manager. The Appendix outlines the library function open as an example to show how the Name Space Manager and Access Manager are tied together. Jade also uses the Sun Shared Library mechanism. By dynamically linking the Jade shared library, most existing software (*ed*, *cc*, *find*, etc.,) are able to access Jade without modification or recompilation.

In comparison with the kernel-approach implementation used by AFS and NFS, the user-level approach has the following advantages: it is easy to experiment with different design options; 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; and the system is more easily ported among different operating systems. A potential disadvantage of this approach, however, is that performance will be degraded by the user-level approach. The next section will address this issue in more detail.

5.2 Performance Results

We measured the performance of the prototype with the Andrew Benchmark developed at CMU[Howa88]. The Benchmark includes five phases: making directories (**MakeDir**), copying files (**Copy**), examining the status of files under one subtree (**ScanDir**), scanning every byte of files under one subtree (**ReadAll**), and compiling files (**Make**).

Two cases are tested: a local area network and a TCP/IP Internet. For each case, we compare the performance of Jade and NFS. For the first case, the physical file system is located on the local area network and a 10 Mbits Ethernet connects this file system and the client workstation. For the

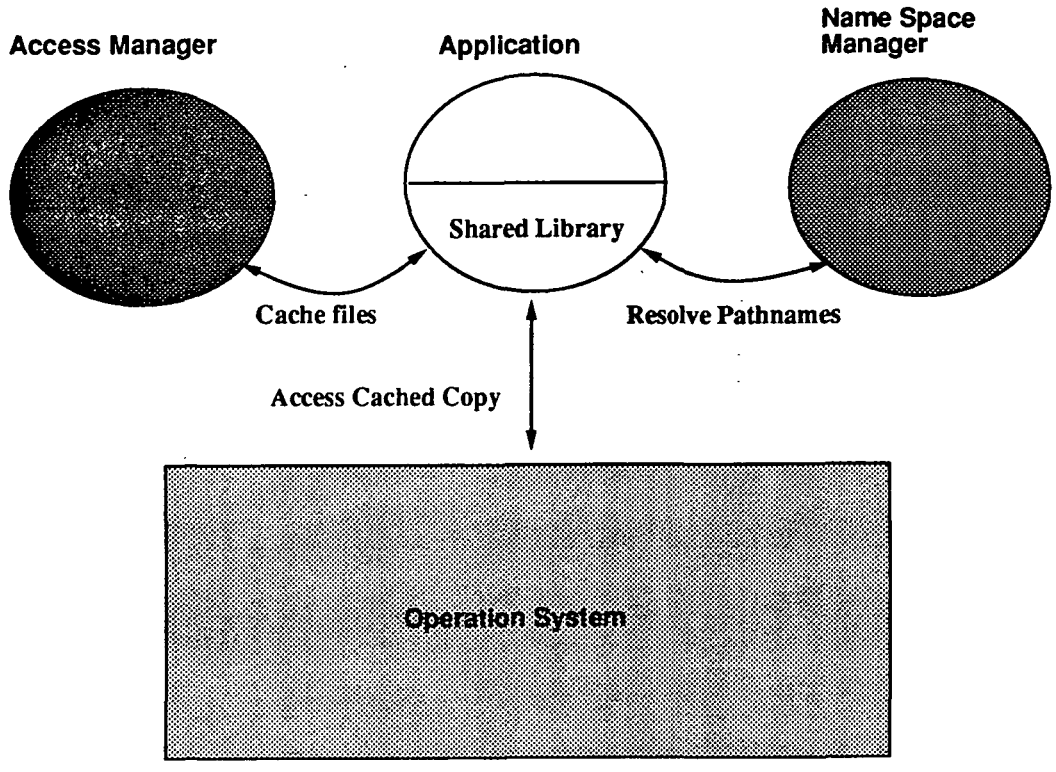

Figure 14: Model of Jade Prototype

Internet case, the file system is located at Purdue University, while the client workstation is at The University of Arizona. There are 24 gateways between the file system and the client workstation, and the communication channels range from 10 Mbits Ethernet to 1.544 Mbits T1 links. The client workstation, where Jade and the Andrew Benchmark are running, is a Sun 4/60 workstation with 16 Mbytes of main memory and a 320 Mbytes local disk. The Access manager uses this local disk to cache files. The client workstation is running Sun OS 4.1. Both file systems provide Sun NFS protocol for file access.

The performance results of both cases are given in Table I. Overall, Jade exhibits a 36% (275 seconds versus 202 seconds) slowdown relative to NFS in the LAN case. We attribute this to the cost of the user-level implementation and indirectly naming in the logical name space. For example, each **open** function needs to consult the Name Space Manager to resolve pathnames[5]. 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 Sprite Pseudo-File-System[Welc89] which shows 33-41% slowdown when running the Andrew Benchmark. In the Internet case, the overall performances of Jade is almost identical to that of NFS, with only a 4% slowdown (1169 seconds versus 1125 seconds). It is clear that the cost of accessing the internet is so high that the penalty of the user-level implementation

[5]In this test case, the consultation to the Access Manager is omitted since files are located on the same host as the Access Manageris located.

is insignificant. Another interesting observation is that the ratio of the LAN case to the Internet case is 4.25 for Jade which is lower than NFS's 5.57.

	LAN		Internet	
	NFS	*Jade*	*NFS*	*Jade*
MakeDir	3 secs	3 secs	23 secs	23 secs
Copy	20 secs	23 secs	299 secs	536 secs
ScanDir	31 secs	52 secs	115 secs	127 secs
ReadAll	50 secs	84 secs	120 secs	139 secs
Make	98 secs	113 secs	568 secs	344 secs
Total	202 secs (1.00)	275 secs (1.36)	1125 secs (1.00)	1169 secs (1.04)

Table I. Performance Results

For the **MakeDir** phase, Jade has the same performance as NFS in both the LAN case and the Internet case. Since no file access is invoked in this phase, this suggests that the performance of pathname resolutions introduced by Jade is at least as good as that used by NFS, or even better if one subtracts the time spent communicating between the client and the Name Space Manager.

For the **Copy** phase, the performance of Jade (23 seconds) is compatible to that of NFS (20 seconds) in the LAN case. However, in the Internet case, the performance of Jade (536 seconds) suffers a 79% slowdown compared with the NFS case (299 seconds). This is because rather than accessing files page by page, Jade caches entire files on the Access Manager. Figure 15 shows 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 call **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. (The performance of the **Copy** phase reported in Table I uses this approach.) The ideal way to solve this problem is to have the access protocol support a new *copy* operation, thereby making the cost of copying files compatible to the cost of *renaming* files.

For the **ScanDir** phase, the performance of Jade in the LAN case is 52 seconds, while NFS is 31 seconds. In the Internet case, the former is 127 seconds and the latter is 115 seconds. The lesson we learn from this phase is about network latency. In a local area network, the network latency is not an issue, and the ratio of the message latency time to the time spent at the client and the server for computation is insignificant. In the Internet, on the other hand, the network latency becomes a major factor in overall performance. 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, these overhead is insignificant. In the Internet, where the network latency is much higher, the

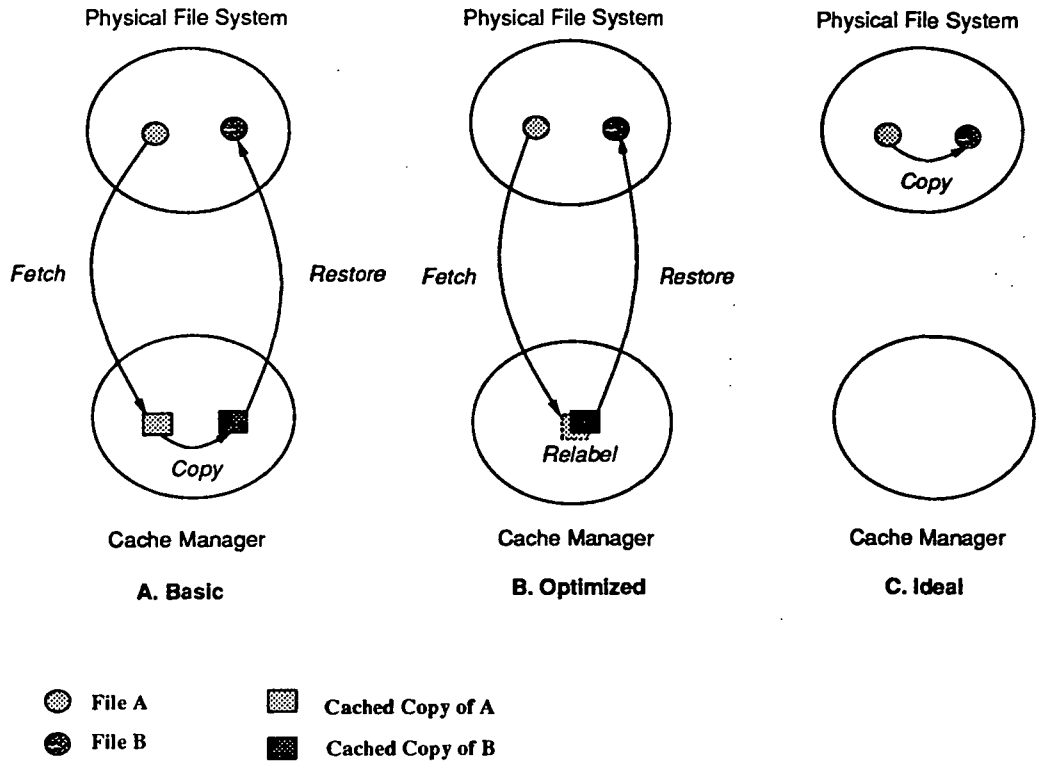

Figure 15: Copy Files

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 original Jade and 38% faster than NFS. Notice the speedup percentage will increases 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 NFS is 50 seconds. For the Internet case, the former is 139 seconds and the latter is 120 seconds. However, when rerunning this phase in the Internet case, NFS, which uses page access, has a similar performance. But 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 is a good example to show that caching entire files is essential in terms of performance for access files in the Internet. In the LAN case, Jade takes 113 seconds and NFS takes 98 seconds. In the Internet case, however, Jade's time drops dramatically to 61% of the NFS's time (344 seconds versus 568 seconds). This result is extremely important since the majority of file access in a research or academic environment invokes viewing, editing, and compiling a small set of files[Floy86].

6 Concluding Remarks

Jade is a client-based, internet-wide file system. Its primary goal is to provide a uniform mechanism to name and access files in an heterogeneous internet environment. Instead of introducing a new file system, we focus on accommodating existing distributed file systems. Because of autonomy in the internet, we restrict our solution to require no modification in software nor change in administration of the existing file systems. We therefore define a logical layer between existing file systems and their users. In order to avoid the complexity of maintaining an internet-wide, global name space, Jade supports a per-user based name space. Finally, Jade allows multiple physical and logical file systems to be mounted in any directory.

We have implemented a prototype of Jade. From experiments using the Andrew Benchmark, we conclude with the following three points. First, Jade's user-level implementation and extra level of indirection cause a 36% slowdown in the local area networks case. However, this cost becomes insignificant compared with the cost to access the Internet, as Jade and NFS have compatible performance in the Internet case. Second, because of the high cost of accessing the Internet, caching entire files is essential. It is particular true when the majority of file access invokes viewing, editing, and compiling of a small set of files. Finally, network latency, which is not an issue in local area networks, becomes an important factor of performance in the Internet. While most of access protocols are design with local area networks in mind, they disclose performance problems from network latency when extended to the Internet. Experiments show that an enhanced *readdir* for obtaining full information of a directory in addition to names speedups the performance about 40% in the **ScanDir** phase of the Andrew Benchmark.

Acknowledgments

Doug Comer and Scott Ballew supported a physical file system at Purdue University for an Internet test. Tyson Henry provided valuable comments on earlier drafts of this paper.

Appendix

This appendix illustrates pseudo codes for the functions described in the paper.

Function of Resolving Path Name

```
ResolvePathName(pathname)
    node := FindClosestAncestor(pathname);
    let remaining_path be the difference between the path of node and pathname;
    let current_path be pathname of node;
    while remaining_path is not empty do
        let component be the first component in remaining_path, and remove it from the path;
        let outstanding_list be the list of references associated with node;
        initiate visiting_list;
        while not found and outstanding_list is not empty then
            let reference be the first element in outstanding_list, and remove it from the list;
            if reference is not found in visited_list then
                add reference into visited_list;
                new_list := Cache(reference);
                new_node := LookupCache(current_path);
                    /* Consult the hash table to get the node with current_path */.
                if new_node is not nil then
                    let node be new_node;
                    let found be true;
                else
                    insert new_list into the front of outstanding_list;
                fi
            fi
        end    /* inner while */
        if not found then
            return nil
        fi
    end    /* outer while */
    return node;
```


Function of Finding Closest Ancestor

```
FindClosestAncestor(pathname)
    while true do
        node := LookupCache(pathname);
            /* Consult the hash table to get the node with pathname */.
        if node is nil then
            remove the last component of pathname;
        else
            if node is a skeleton node then
                return node;
            else      /* The node is a cached node. */
                check the validity of node ;
                if node is valid then
                    return node;
                else
                    remove the last component of pathname;
                fi
            fi
        fi
    end
```

Function of Listing Directory Entries

```
Dir(pathname)
    dir_node := ResolvePathname(pathname);
    if dir_node is nil then
        return fail;
    children := LookupCache(SCT, pathname);
    let outstanding_list be the list of references associated with dir_node;
    initiate visiting_list;
    while outstanding_list is not empty do
        let reference be the first element in outstanding_list and
            remove it from the outstanding_list;
        if reference is not found in visited_list then
            add reference into visited_list;
            (new_list, new_children) := Cache(reference);
            children := children ∪ new_children;
            insert new_list into the front of outstanding_list;
        fi;
    end;      /* while */
    return children
```


Open Function

```
open(path, flags, mode)
    if path is not Jade filename then
        return syscall(SYS_open, path, flags, mode);
    fi

    if open for read then
        file_ref := NameSpaceManager.Resolve(path);
        if file_ref is not nil then
            cache_ref := AccessManager.Fetch(file_ref);
            if cache_ref is not nil then
                return syscall(SYS_open, cache_ref, flags, mode);
            fi
        fi
        if open for read only then
            return fail;
        fi
    fi

    if open for write then
        let dir be path's directory;
        dir_ref := NameSpaceManager.Resolve(dir);
        if dir_ref is not nil then
            cache_ref := AccessManager.NewCache(dir_ref, path);
            return syscall(SYS_open, cache_ref, flags, mode);
        fi
    fi
    return fail;
```

References

[Allm86] Allman, E. An introduction to the source code control system. In *Unix Programmer's Manual Supplementary Documents Volume 1.* University of California at Berkeley, April 1986.

[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.

[Come85] Comer, D. and Murtagh, T. P. The Tilde File Naming Scheme. *IEEE Transactions on Software Engineering*, pages 509–514, 1985.

[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. Technical Report TR 177, Computer Science Department, The University of Rochester, March 1986.

[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.

[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.

[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.

[Pete91] Peterson, L. L. and Rao, H. C. Software development in the Jade file system. Technical report, Department of Computer Science, University of Arizona, 1991. In preparation.

[Pike90] Pike, R., Presotto, D., Thompson, K., and Trickey, H. Plan 9 from Bell Labs. In *Proceedings of the United Kingdom Unix Users Group*, London, England, July 1990.

[Pope85] Popek, G. J. and Walker, B. J. *The LOCUS Distributed System Architecture.* The MIT Press, 1985.

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

[Ritc78] Ritchie, D. M. and Thompson, K. The UNIX Time-Sharing System. *Bell System Technical Journal*, 57(6), July 1978.

[Saty85] Satyanarayanan, M. The ITC Distributed File System: Principles and Design. In *Proceedings of the Tenth ACM Symposium on Operating System Principles*, pages 35–50, December 1985.

[Saty89] Satyanarayanan, M. Distributed File Systems. In *Distributed Systems*, pages 149–188. ACM Press, 1989.

[Saty90] Satyanarayanan, M. Scalable, secure, and highly available distributed file access. *IEEE Computer*, 23(5):9–21, May 1990.

[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.

[Sun86] Sun Microsystems, Inc., Mountain view, Calif. *Network File System*, February 1986.

[Terr87] Terry, D. B. Caching hints in distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):48–54, January 1987.

[USC85] USC, . File Transfer Protocol (FTP). Request For Comments 959, USC Information Sciences Institute, Marina del Ray, Calif., October 1985.

[Welc86] Welch, B. B. and Ousterhout, J. 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. 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, Information Technology Center, Carnegie Mellon University, August 1988.