THE JADE FILE SYSTEM

(Ph.D. Dissertation)

Herman Chung-Hwa Rao

TR 91-18

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF
ARIZONA
TUCSON ARIZONA

THE JADE FILE SYSTEM

(Ph.D. Dissertation)

Herman Chung-Hwa Rao

TR 91-18

August 15, 1991

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

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.

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the copyright holder.

SIGNED: _____

ACKNOWLEDGMENTS

I am deeply indebted to my advisor, Larry Peterson. His guidance and example as a scientist have directed my development in the field of computer science, and his patience and support have made this work possible.

I am also grateful to other members of my committee, Richard Schlichting and Scott Hudson, for their comments and suggestions on my work. I thank the minor members of my committee, Fredrick Hill and Ralph Martinez, for helping with my graduate program. I also thank Richard Snodgrass for his inspiration.

I want to thank my fellow graduate students, Tyson Henry, Sun Wu, Vic Thomas, Andrey Yeatts, Shamim Mohamed, Nick Kline, Shivakant Mishra, Patrick Homer, Edwin Menze, Jim Knight, and Naiwei Lin, for their friendship. Particularly, I am indebted to Tyson Henry, who read the draft of this thesis and gave valuable comments.

Finally, I would like to thank my parents for their faith and support; my wife, Zoe, for her patience and love, and being my best friend; my daughter, Stephanie, for the happiness we had together; and my sisters for their encourage.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

File systems have long been the most important and most widely used form of shared permanent storage. File systems in traditional time-sharing systems such as Unix support a coherent sharing model for multiple users. Distributed file systems implement this sharing model in local area networks. However, most distributed file systems fail to scale from local area networks to an internet. This thesis recognizes four characteristics of scalability: size, wide area, autonomy, and heterogeneity. Owing to size and wide area, techniques such as broadcasting, central control, and central resources, which are widely adopted by local area network 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 independent organizations. Finally, heterogeneity is the nature of an internet file system, not only because of its size, but also because of the autonomy of the organizations in an internet.

This thesis introduces the Jade File System, which provides a uniform way to name and access files in the internet environment. Jade is a logical system that integrates a heterogeneous collection of existing file systems, where heterogeneous means 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. In Jade's design, we pay careful attention to avoiding unnecessary network messages between clients and file servers in order to achieve acceptable performance. 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.

CHAPTER 1

INTRODUCTION

Internets, such as the National Research and Education Network (NREN)[Come91], provide an opportunity to unite geographically dispersed users and computing resources into an integrated computing environment. They allow users throughout the country to exchange mail, share files, and access databases and supercomputers. As networking facilities become increasingly ubiquitous, the number of available resources can be expected to grow by several orders of magnitude. Software infrastructures that support access to, and sharing of, these resources must also be upgraded to take advantage of this connectivity. The File Transfer Protocol (FTP)[Post85], the TELNET protocol[Post83], and the Simple Mail Transfer Protocol (SMTP)[Post82], all designed more than a decade ago, are still the primary services and tools by which users take advantage of the internet[Cace91]. A lack of mechanisms for transparently naming and accessing resources limits the utilization of internet resources.

This dissertation presents a new distributed file system, called Jade, that addresses internet-wide resource sharing in the context of file systems. Jade is both an infrastructure for accessing files in an internet and a framework for collaboration among geographically dispersed users. It provides a uniform mechanism to name and access files located in the internet—remote files are named and manipulated in the same way as local files.

1.1 Need for an Internet File System

File systems have been the most important and most widely used form of shared permanent storage. A file system provides users with the abstraction of a file, thereby freeing them from concerns about the details of storage locations and disk allocations. File systems in traditional time-sharing systems such as Unix[Bach86][Leff89] support a coherent sharing model in which multiple users share files. Distributed file systems such as the Locus Dis-

tributed System[Pope85][Walk83], the Sprite File System[Nels88][Welc86], Sun's Network File System[Sand85][Sun86a], and the Andrew File System[Howa88][Saty85] implement this sharing model in local area networks.

An internet-wide file system further distributes this sharing model across multiple autonomous sites that span an entire internet. Such a file system allows users to access files located anywhere in an internet, and serves as a sharing mechanism for geographically dispersed users. Although it is difficult to characterize precisely—since no wide-spread internet file system currently exists—we anticipate that the availability of an internet file system would have three major impacts on the internet community. First, it would support access to a variety of resources. Second, it would encourage inter-organization collaboration. Finally, it would increase user mobility. Consider each of these three impacts in turn.

Accessing a Variety of Resources

An internet contains a rich variety of resources, including file systems, databases, information archives, supercomputers, and specialized hardware. The *Internet Resource Guide*[NSF89] reports scientific resources available in the internet, including computational resources, library catalogs, and biological and software data archives. For example, *Request for Comments* (RFCs)[Come91], technical reports, public domain software, and information archives are accessible through the internet. As another example, NSFNET[Come91] gives scientists access to supercomputers located at various Supercomputer Centers (e.g., Pittsburgh Supercomputer Center, San Diego Supercomputer Center, etc.).

Unfortunately, adequate user-level facilities to access the remote resources are not yet available. There is a need to give the internet community the ability to make effective use of the internet in accessing the remote resources[Lein87]. File systems have been proven to provide an effective framework for accessing resources in time-sharing systems and local area networks. It is reasonable to expect that file systems in the internet would provide users with a consistent, effective, and transparent way to name and access remote resources.

Consider the following scenario for accessing a supercomputer. Users prepare data

files and programs at their local workstations, transfer all the files needed from the local environment to the supercomputer environment using FTP[Post85], and then execute the task on the supercomputer. After the task is finished, they retrieve results over the network, again by FTP. Typically, users repeat this cycle several times until the programs and data are correct. In order to transfer files properly, users need to know the details of FTP, as well as of the networks; this is not easy for naive users. With the sharing model provided by an internet file system, however, users would be able to invoke a job on a supercomputer located in the internet in the same way they do on local area networks.

Encouraging Inter-organization Collaboration

Using an internet file system as a sharing mechanism, we expect new collaborations to arise, because geographical distance among members will no longer constrain the carrying out of tasks or sharing of data. Moreover, the ease with which members in a collaboration can exchange their designs, data, images, and documentation contributes in large part to the ease with which those members can share their ideas and knowledge.

Within research and academic environments, inter-organization collaboration is growing rapidly. The report *Towards a National Collaboratory*[Lede89] introduces the concept of the National Collaboratory and states that such a collaboratory for the scientific community

> *will significantly increase the productivity of science and engineering, accelerate the pace of discovery, and amplify the capabilities of human intellect.*

A scalable, internet-wide file system would be the first step toward the goal of a National Collaboratory. This is because applications for sharing resources across the nation would be much easier to build on top of such a file system than from scratch.

Increasing User Mobility

An internet file system would significantly increase user mobility. At present, there is an explosion in the number of computers in the world. There are supercomputers in national supercomputer centers, large mainframes in campus computer centers, minicomputers for

departmental use, workstations used by individuals, and personal computers at home for personal use. Many people use more than one computer; for example, a personal computer for handling personal finance, a workstation for writing papers and documents, a mainframe for heavy computing, supercomputers for large dynamic simulations, and remote computers for accessing specialized databases. That is, the work space of an individual user consists of multiple machines spread across multiple administrative boundaries. It is also important that people be able to work effectively while at different offices or when traveling. An internet file system would be able to present users with a homogeneous view of these heterogeneous machines. Moreover, with such a file system, the user would be able to incorporate files located on different machines in order to execute a task. For example, with the same command as is used for local files, the user could include simulation results from the supercomputer when editing a paper from a workstation.

1.2 Problem: Scalability

Many distributed file system designs have been proposed and implemented over the last decade. Issues concerning the design of distributed file systems, including naming and transparency, consistency and availability, remote access methods, and fault tolerance, are well documented in several survey papers[Levy90][Saty89a][Svob84]. Designing an internet-wide file system, however, introduces a new orthogonal issue: *scalability*. In this section, we describe the problems raised when one scales a file system up to an internet.

We envision an internet file system encompassing millions of participants, where by participants we are referring to servers maintaining resources, and clients consuming resources. Such a file system would have four critical characteristics:

- Size—number of participants;
- Area—geographical distance among these participants;
- Autonomy—independence and self-determination of individual participants;
- Heterogeneity—diversity of software and hardware of participants.

Consider these four characteristics in turn.

Size

The number of participants in a large internet is vast. Comer has documented that in 1990, the connected internet included more than 3,000 active networks and 200,000 computers at universities, government agencies, and corporate research laboratories[Come91]. An even more important property is the *evolutionary growth* of the environment. Comer estimates that in late 1987, the growth of the internet had reached 15% per month[Come91].

A distributed system based on static assumptions that the number of participants is bounded by a constant will not scale well. Barak and Kornatzky[Bara87] express this point from a different perspective:

> *The service demand from any component of the system should be bounded by a constant. This constant is independent of the number of nodes in the system.*

That is, any service mechanism whose load demand is proportional to the size of the system is destined to break down once the system grows beyond a certain size. This principle can be applied to channels and network traffic, and hence prohibits the use of broadcasting, which is an activity that involves every server in the network. The large size also prevents participants from attempting to maintain information about the global state of the system.

Wide Area

In an internet file system, participants may be located anywhere in the internet. A wide area, in contrast to a local area, presents another property of the scalability problem: *distance*. Because of distance, the cost to access resources located in the internet becomes significant in comparison with the cost to access resources in local area networks. More precisely, in local area networks, network latency is not an issue, and the ratio of the message latency time to the time spent at hosts for computation is insignificant. In a wide area network, on other hand, network latency becomes a major factor in overall performance. For example, in the current NSFNET, it takes about 100 milliseconds to send a packet round trip to a nearby site, and about 400 milliseconds to a site across the country[Scha90]. On the other hand, the user-to-user round trip from a host to another

connected by a 10Mbps Ethernet is about 2 milliseconds[Hutc89a]. Avoiding unnecessary network messages between participants is critical to performance. Therefore, well-designed caching mechanisms and proper communication paradigms are very important in the design of an internet file system.

Autonomy

Although a large distributed system could be built from scratch, an internet-wide file system must be created by joining together a collection of existing, independent distributed file systems. This is because a variety of distributed file systems have been widely used by distinct, autonomous organizations[Cabr88][Howa88][Nels88][Pope85], and it is unlikely that any single file system will ever be universally accepted in an internet environment. Indeed, distribution of these existing systems over an internet is the consequence of communication advance rather than the result of a dedicated design. In a system composed of a multitude of cooperating participants, the *autonomy* of each entity must be respected in order to accommodate each of the separate participants to form the system. In fact, the property of autonomy is inherited from the internet itself: The connectivity of an internet is based on the willingness of individual autonomous organizations to participate in a shared environment. Consequently, the architecture of an internet file system must support the notion of autonomy in order to scale well in practice.

Heterogeneity

A large-scale system also implies a high degree of heterogeneity, in both hardware and software. This is particularly true because an internet file system consists of a collection of autonomous organizations, each of which has the freedom to install and use its own systems according to its own internal needs. How to choose an appropriate layer to accommodate heterogeneity is a major task in designing an internet file system. For example, portability and heterogeneity contribute to the popularity of the Sun Network File System[Levy90][Saty89a]. To facilitate portability and accommodate heterogeneity, NFS distinguishes between the access protocol and the implementation of the file system.

1.3 Solution: The Jade File System

This dissertation proposes the Jade file system as a solution to the problems of scalability. Jade has the following characteristics:

- It integrates existing, heterogeneous distributed file systems.
- No modification in software or change in administration of the underlying file systems is necessary.
- It provides a per-user logical name space.
- It facilitates sharing by mounting logical name spaces.
- It allows multiple file systems to be mounted into one directory.
- It caches entire files on disk.

The following discusses each of these characteristics, focusing on how Jade addresses problems identified in the previous section.

Jade is a logical system that integrates a heterogeneous collection of existing file systems. It does 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 for communications between file servers and their users. Examples of access protocols include the protocol used by Sun's Network File System, the protocol defined by the Andrew File System, and the File Transfer Protocol. The access protocol not only provides the key to access file systems, but also hides the heterogeneity of the operating systems and the architectures of the hosts where file systems are located.

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 instance of the Jade File System as a regular file system user without any special privileges.

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 heterogeneous, internet-wide file systems, regardless of what machine he or she is using. A global name space for a time-sharing system is one of the major features provided by Multics[Orga72][Salt78] and Unix[Ritc78].

Most of the distributed file systems have inherited this idea and support global name spaces for local area networks or for campus-wide networks. However, maintaining a consistent and coherent global name space for a large internet is not a trivial task. Jade trades the burden on an administrator of maintaining a global name space for the burden on the user of organizing a private name space. Hence, the complexity of Jade is bounded by the number of files accessed by one user.

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 a Jade file system. 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.* With the multiple mount, users are able to group files from different file systems under one directory and transparently locate files replicated on several file systems. Moreover, a set of users are able to share a collection of files stored on different file systems without worrying about interference from one another. To take advantage of this feature, we have developed a new software development environment on top of Jade.

Jade employs whole file caching. Opening a file causes it to be cached in its entirety, on some nearby disk. Reads and writes are directed to the cached copy without involving the original servers. The valid cached copy can be used for further opens as well. Because of the high cost of accessing remote servers, complete file caching is needed to reduce the network traffic; it is essential for good performance in the internet.

1.4 Overview of Related Work

This dissertation is related to research in two broad, and perhaps overlapping, areas: distributed file systems and naming systems. In summary, most distributed file systems commit to a single access protocol and are designed for local area networks[Saty89b]; they do not scale well over a large internet. Many naming systems, on the other hand, are

designed for large internets. However, they are mainly used to map objects other than files (e.g., mailboxes and hosts), and it is too expensive to invoke these naming systems whenever a file is opened[Cher89].

1.4.1 Distributed File Systems

Countless distributed file systems have been developed over the last decade, and many of the well-known efforts are surveyed by Svobodova[Svob84], Satyanarayanan[Saty89b], and Levy and Silberschatz[Levy90]. This section summarizes this work, with an emphasis on how these systems do not scale in one or more of the four dimensions outlined in Section 1.3.

Locus[Pope85] and Sprite[Nels88] are designed to integrate several Unix machines into a single virtual machine. The Locus Distributed System provides facilities for file replication and location transparency. In order to maintain consistency among replicated copies, there is a *current synchronization site* associated with each file group. Such a central controller of an otherwise distributed mechanism becomes a performance bottleneck when the system grows beyond a certain size. 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 mentioned before, broadcasting invokes *every* node in the network, making it unrealistic in a loosely coupled environment such as an internet.

The Amoeba File System[Mull85][Tane90] examines the concept of layered file services and uses a *central* directory server to map a string name of an object into its capability. As mentioned before, this central server would be a performance bottleneck when the system becomes large scale. This is particularly true for the directory server because profiling studies for Unix-like systems show that nearly one-quarter of the time in the kernel is spent in the pathname translation[Leff89]. Levy and Silberschatz have pointed out that centralization is a form of functional asymmetry between components composing the system, and central control schemes should not be used to build scalable systems[Levy90].

Sun Microsystems' Network File System (NFS)[Sand85][Sun86a] and CMU's Andrew File System (AFS)[Howa88][Saty85] do not require all participant machines to be tightly

connected as in Locus or Sprite. Both systems support the concept of a global name space and provide mechanisms to build such a name space. In order to build a global name space, NFS requires all the hosts to mount each other's file systems. 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. Scalability is the dominant design consideration in the Andrew File System. However, the original design[Howa88][Saty85] considers 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. Thus, autonomy is the major problem of this design. It has proven very difficult to persuade each autonomous site to give up the file systems currently running, and switch to the Andrew File System. Furthermore, one common drawback of the global name space approach, from users' perspective, is that pathnames in the global name space become longer. It becomes increasingly difficult to search for files whose names are not precisely known. For example, a typical home directory for the user John is "**/afs/cs.arizona.edu/usr/john**" in the Andrew File System. In contrast, one could consider this the root ("/") of John's personal file system.

Projects such as Tilde[Come86][Come85], QuickSilver[Cabr88], and Plan 9[Pres91][Pike90] provide mechanisms to let users construct their own name spaces rather than a single global name space. 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 all commit to a single protocol-suite. Tilde allows users to choose individual name spaces (called *trees*) to form their naming environment (called a *forest*). However, it does not allow one tree to 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.

1.4.2 Naming Systems

A great deal of effort has been spent on the design and implementation of a global naming facility. Perhaps the most advanced work of this kind are Grapevine[Birr82], Lampson's Global Name Space[Lamp86], and the Domain Name Service[Mock87] of the DARPA internet. Work in this area is surveyed by Terry[Terr85]. Although these naming systems are designed for large-scale environments, they are mainly used to resolve names for hosts, mailboxes, or network services, and they are practically applicable only to objects that do not need to be looked up frequently. It is too expensive to invoke the global name service whenever a file is opened[Cher89].

Cheriton and Mann[Mann87][Cher89] extend these global naming designs and focus on issues of performance and fault tolerance. This system is based on the fundamental requirement that each server knows the *full* global names of the objects it maintains. Therefore, the structure of the global name space is rigid and requires the full cooperation of each participating name server. Another problem of this design is the availability of multicast in the internet, which is used to locate name servers. While being supported by some local area networks, the multicast is not generally available in the internet.

1.5 Thesis

The major thesis advanced by this dissertation can be stated as follows:

> *An internet-wide file system that is scalable and has acceptable performance can be built by integrating heterogeneous file access protocols, and by providing users with their own private name spaces.*

In supporting this thesis, the dissertation makes two contributions:

- We have designed, implemented, and evaluated an internet-wide file system that provides each user with a completely homogeneous view of heterogeneous file systems.
- We have invented a rich set of naming facilities, including per-user logical name spaces, mounting logical name spaces, multiple mounts, name space stacks, and a

generalization of a symbolic link and a directory. These naming facilities not only are useful to access internet files, but also are applicable to a variety of applications.

This dissertation is structured as follows. Chapter 2 justifies the design of the Jade file system. It describes how to construct a logical file system using existing heterogeneous physical file systems as building blocks. It concludes with an overview of Jade's structure, including its two major components: a Name Space Manager and an Access Manager.

Chapter 3 presents the Name Space Manager that maintains the name space. It describes its two novel features: allowing multiple file systems to be mounted under one directory, and permitting one logical name space to be mounted in another logical name space. Because of these features, the semantics of directories in Jade differs from those in other Unix-like file systems. It also makes pathname resolution more complicated in Jade than in other file systems.

Chapter 4 describes the Access Manager that supports access to files located on the internet. This chapter delineates the design of the Access Manager and discusses its caching scheme. Jade allows users to choose any available physical file system as the cache server, and it caches entire files on that server. In order to reduce the number of messages exchanging between the cache server and the underlying file systems, the Access Manager implements two delayed-write policies: *write-on-close* and *create-on-close*.

Chapter 5 evaluates the design of the Jade file system. After describing the implementation of the prototype—which consists of the interface to the access protocols UFS, NFS, and FTP—this chapter reports the performance of the prototype using the Andrew Benchmark[Howa88]. It concludes by re-examining design issues based on experience with Jade, emphasizing the tradeoffs of alternative choices.

Chapter 6 describes applications of the Jade file system. In addition to examining Jade's features from an application perspective, the chapter presents several examples that illustrate how one takes advantage of these unique features.

Chapter 7 summarizes the contributions and suggests future research. Issues of consistency control are not addressed in this thesis. Because of the autonomy restriction, Jade does not modify heterogeneous access protocols, and therefore inherits consistency problems from them. In order to support more complicated applications, there is a need

for a more sophisticated control mechanism. Transarc's DEcorum File System[Kaza90] suggests a token mechanism to preserve single-system Unix semantics. The Coda file system, developed at Carnegie Mellon University, focuses on issues of availability in the face of server and networks failures[Saty90b]. Chapter 7 suggests directions for future research in this area.

CHAPTER 2

DESIGN OVERVIEW

This chapter motivates and justifies the design of the Jade file system. First, it abstractly defines the underlying physical file systems upon which the Jade file system is built. Second, it introduces Jade's salient features, focusing on its logical name space. Jade's name space provides users with two novel features: It allows multiple file systems to be mounted under one directory, and it permits one logical name space to be mounted in another logical name space. The chapter concludes with an overview of Jade's implementation structure, including its two major components: a Name Space Manager and an Access Manager. The following two chapters explore these two components in more detail.

2.1 Physical File Systems

Abstractly, Jade adopts a very 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), the protocol defined by the Andrew File System (AFS), and the File Transfer Protocol (FTP). 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). For the purpose of this thesis, 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

correspondingly, not the file system itself. A given physical file system may adopt one or more access protocols for remote access. For example, many physical file systems support a default access protocol (e.g., AFS or NFS) and provide FTP as an additional service. The access protocol not only provides the key to accessing the physical file system, but also hides the heterogeneity—both in hardware and in software—of the system on which the physical file system is located. That is, once the proper access protocol is available, it should be 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. This interface acts like a switch among these heterogeneous access protocols and maps operations defined by the interface into functions provided by distinct access protocols. This interface consists of the following operations:

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.

A complete description of the interface, and how it is mapped into common access protocols (i.e., UFS, NFS, AFS, and FTP), is given in Chapter 5.

Notice that the uniform interface separates the directory operations (GetEntries, RemoveEntry, and GetAttr) completely from the file access operations (Fetch and Restore). The significance of this is that the Fetch operation is not used to access directories in a physical file system; the GetEntries operation is used instead. The reason behind this

separation is based on the observation that the *directory* abstraction is different in dissimilar file systems. In fact, most file access protocols provide a completely different set of operations for directory manipulation.

Consider a physical file system maintained by a host with the domain name[Mock87] **meg.cs.arizona.edu**. The host supports the access protocol NFS to access this physical file system identified as **/usr** by the host. Hence, from Jade's point of view, this physical file system is described by the triple

<NFS, meg.cs.arizona.edu, /usr>

When the access protocol and the exact host-specific identifier are not germane to the discussion, we use the shorthand notation:

Server_Name:Pathname

to refer to the physical file system. In this example, **meg:/usr** specifies the file system. As mentioned before, operations defined by the uniform interface are mapped into the corresponding functions supported by the access protocol. In this example, the Fetch (Restore) operation is realized by a sequence of NFS's read (write) operations[Sand85][Sun86a], while the GetEntry, RemoteEntry, and GetAttr operations are directly mapped to NFS's readdir, remove, and getattr operations, respectively.

2.2 Logical Name Space

This section describes the logical name space that is central to the Jade file system. After discussing the name space in general, it illustrates how one might build a logical name space from a collection of heterogeneous physical file systems. Like other Unix-like distributed file systems[Saty85][Sun86a][Welc86], Jade supports the mount mechanism to glue individual file systems together. However, this mechanism is more complicated in Jade than in other systems and requires more explanation. The section completes the picture by discussing the relationship among a collection of logical name spaces in the internet.

2.2.1 Per-User Name Space

Like most Unix-like file systems[Ritc78][Saty85][Welc86], 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. Figure 2.1 illustrates an example of a Jade file system associated with the user John. 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.

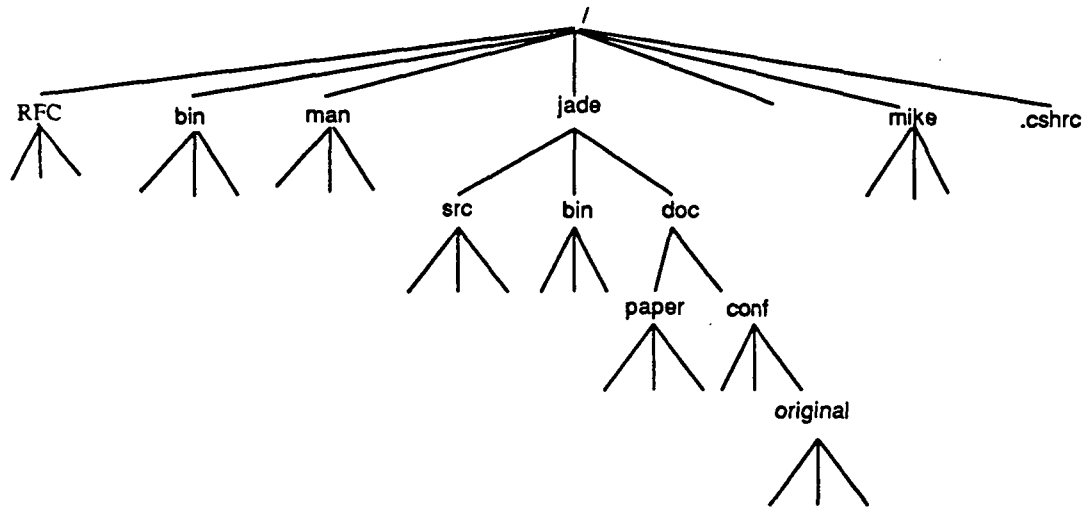


Figure 2.1: Logical Name Space

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 system 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 such as X Window[Sche86] and Sun NeWS[Gros86] 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[Cabr88][Floy86b][Shel86]. Satyanarayanan[Saty89a] 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 administrators of maintaining a global name space for the burden on users of organizing private name spaces. 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's file system, 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 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.

As mentioned before, pathname resolution is performed in the context of a specific user's 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, 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: 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. The attached name space can also be a special name space defined for only a program. For example, a front-end program of a database system can define its own name space to match internal file organizations.

2.2.2 Building a Logical 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 2.2 illustrates a possible partitioning of the logical name space shown in Figure 2.1.

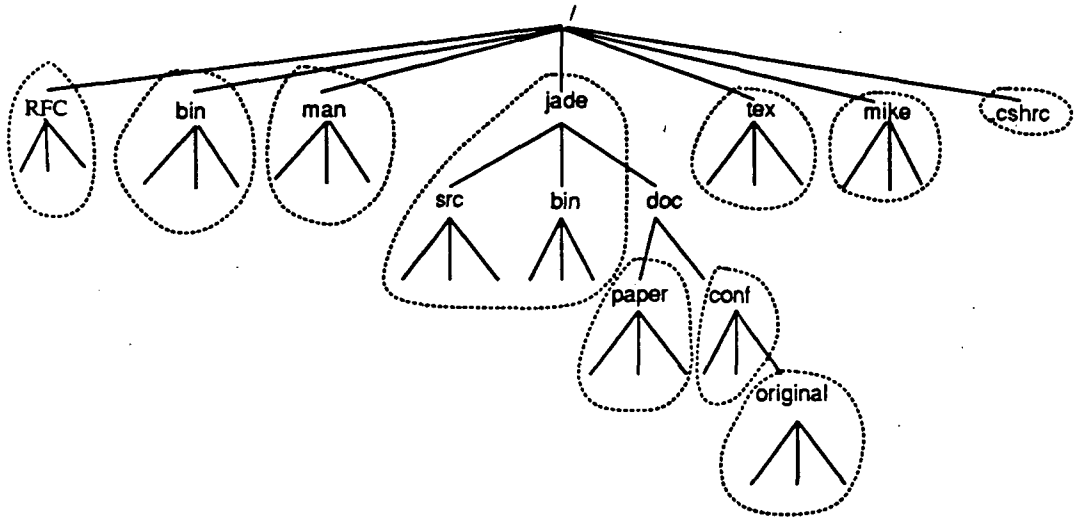


Figure 2.2: Partition of a Logical Name Space

Like most distributed file systems, Jade supports a *mount* operation that is used to attach 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. It is only a skeleton because most of the files/directories within a given domain are maintained by some physical file system, not by Jade. Only the roots of mounted file systems, called skeleton directo-

ries, are maintained by Jade. Another way of saying this is that the skeleton directories are superimposed over a collection of existing file hierarchies; much of the structure of the underlying hierarchies remains visible to the user.

2.2.3 More about the Mount Operation

Traditionally, the *mount* operation attaches a physical file system to a leaf of the existing naming hierarchy. Jade enhances the mount operation in two ways. First, the mounted file system can be either a physical file system or another logical file system. Second, Jade allows more than one file system to be mounted on one directory.

In order 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 2.2, for example, the directory **/mike** refers to a logical name space belonging to another user, Mike. Simply by concatenating the prefix **/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. In general, this indirect naming can be of arbitrary depth, in that a sequence of logical name spaces needs to be searched in order to locate a desired file. Chapter 3 discusses issues raised in this search and introduces the algorithm we use.

Not only does this feature support file sharing, it also encourages users to generate *auxiliary* name spaces for special purposes. To help manage these auxiliary name spaces, Jade introduces the idea of a *Name Space Stack*. A Name Space Stack is a stack of name spaces owned by a user. The top name space in the stack is the only one that is accessible from outside of the stack. However, every name space in the stack can mount name spaces underneath. By analogy, each name space is like a *translucent paper* that may either hide information below it or contain new information. The view provided by a Name Space Stack is the view of a stack of overlapping translucent papers. The Name Space Stack also provides a simple way to perform *checkpoint* and *rollback* on mount operations. The concept of Name Space Stack is discussed in the next chapter.

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 2.2, for example, the directory **/bin** might refer to three physical file systems: **/usr/john/bin** in the host **jag**, **/usr/john/bin** in the host **meg**, and **/usr/bin** in the host **meg**. Entries under the logical directory **/bin** include those from these three physical file systems. As another example, the directory **/RFC** keeps documents of *Request For Comments* (RFC) distributed by the Network Information Center, and it refers to two physical file systems: **meg:/usr/john/RFC**, where some local, cached copies of RFCs are located, and **nic.ddn.mil:RFC**, where the original files are stored.

This feature, called a *multiple mount*, has a number of advantages, especially when compared with auxiliary mechanisms built on top of other file systems. 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 using 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, where they can be used to handle version control and software distribution. Chapter 6 describes these applications in more detail.

2.2.4 Confederation of Logical Name Spaces

By mounting logical file systems, users can name and access files through other logical name spaces. This indirect naming can be of arbitrary depth and is completely transparent to the user. Figure 2.3 depicts the relationship between one logical name space and a collection of other physical and logical name spaces.

By mounting logical name spaces, these individual logical name spaces are *linked* to each other to form a loosely coupled *confederation*. This confederation has two important characteristics. First, Jade does not enforce specific configurations, or any kind of naming conventions on underlying physical file systems. Instead, each physical file system is viewed as a *building block* with a uniform interface. Users can construct their name spaces with these building blocks according to their own conventions and preferences. Second, Jade

does not require a central authority to organize and administrate individual logical name spaces. Instead, each logical name space is considered as an *autonomous* unit. That is, administration of name spaces is fully *decentralized*. The relationship among all logical name spaces is arbitrary and voluntary.

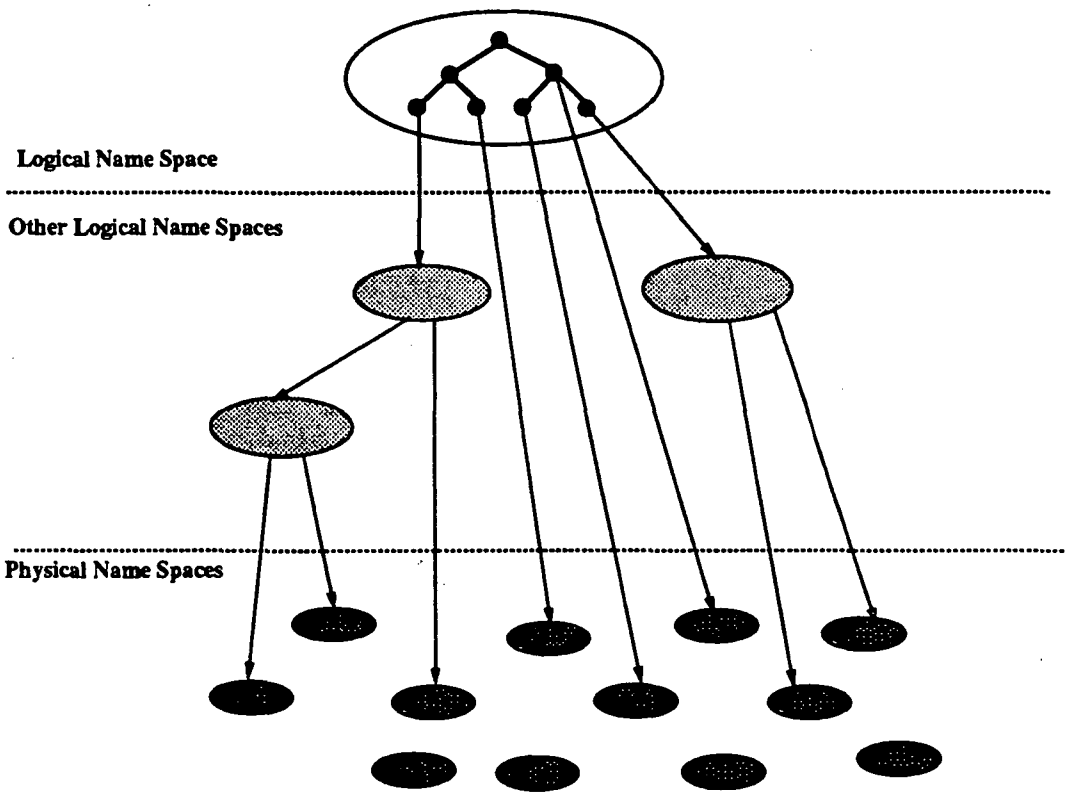


Figure 2.3: Relationship Among Logical and Physical Name Spaces

It is worth noting that it is still possible to build a global, internet-wide name space on top of Jade without any modification to the file system. A logical name space that includes only other logical name spaces is called the *backbone* name space. Chapter 6 presents one example that uses backbone name spaces to construct a global naming environment.

2.3 System Structure

The Jade file system provides a logical layer between existing file systems and their users. It consists of two major pieces: a Name Space Manager and an Access Manager. The

Name Space Manager provides a directory service that maps a logical file name provided by the user into a *file reference*; it is called when opening files. A file reference consists of the address of the physical file system where the desired file is located, the name of the protocol used to access the physical file system, and a handle used by the physical file system to identify the desired file. Given a file reference, the Access Manager supports file access by caching entire files on a nearby physical file system.

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 are built on top of the uniform interface described in Section 2.1, depending indirectly on the underlying file access protocols. For example, a Jade file system might depend on NFS, AFS, FTP, and UFS, as illustrated in Figure 2.4.

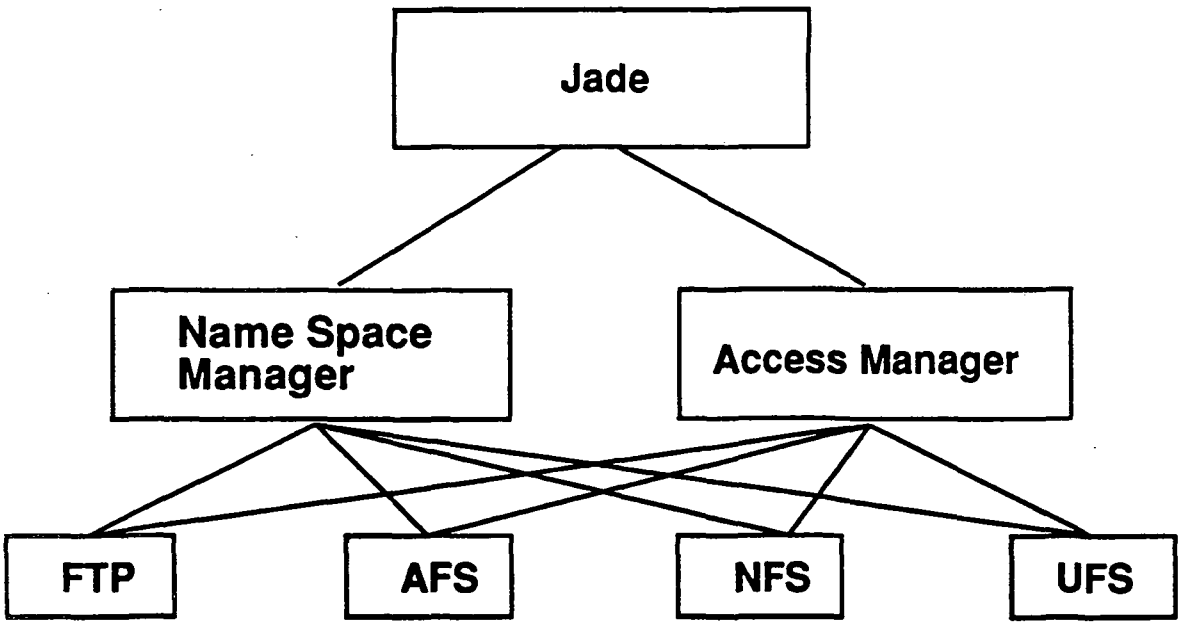


Figure 2.4: Relationship Between Jade and Other File Access Protocols

2.3.1 Name Space Manager

The Name Space Manager implements a logical name space by maintaining skeleton directories that keep track of the boundaries between the underlying file systems. Each skeleton

directory maintains a list of references to other file systems. The mount operation is used to attach a given file system to the name space by creating a new skeleton directory. Jade generalizes the mount operation to allow none, one, or more file systems to be attached to a single skeleton directory. 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 directory, and then resolves the rest of the pathname by consulting the underlying file systems referred to by references stored in the skeleton directory. The underlying file systems are queried using the GetEntries operation defined by the uniform interface. However, 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.

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 the Network File System[Sand85], 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. 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 directory and maintains it only at 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 directory 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. Because the name space is defined on a 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.

2.3.2 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 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.

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 major advantage of this refinement 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. Furthermore, this cache server is actually a secondary cache; the operating system of the workstation has its own in-memory caching

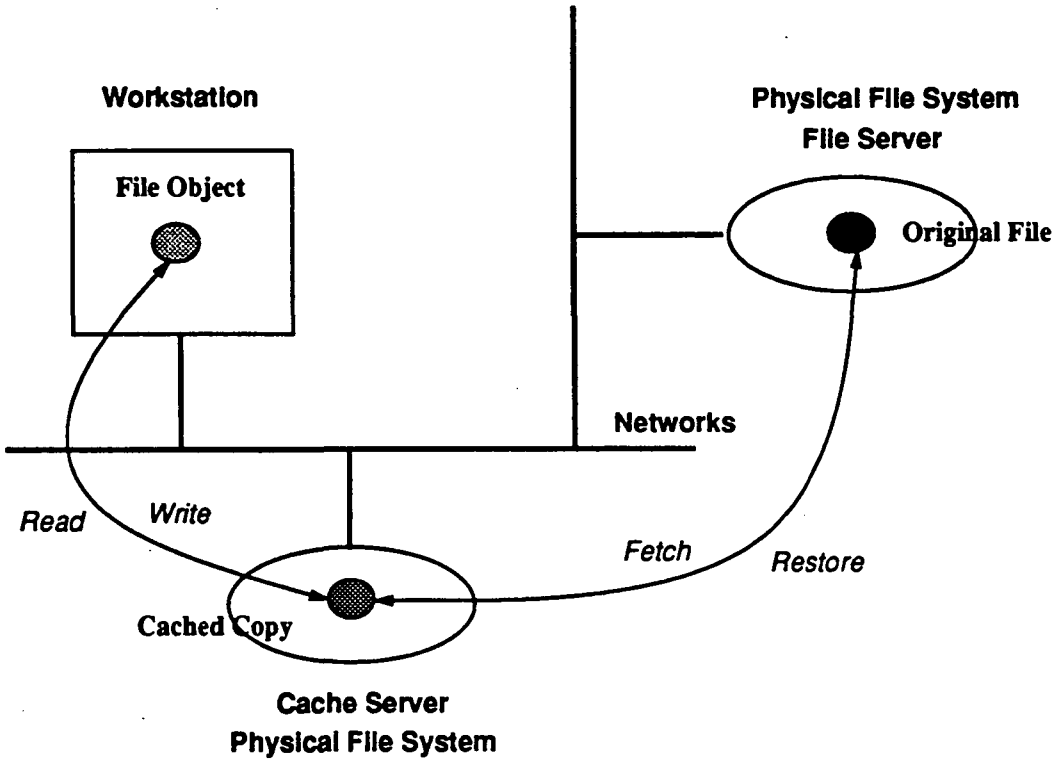


Figure 2.5: File Access

mechanism for accessing files on the cache server. Therefore, it is possible to implement a new cache policy on the cache server. This policy is designed specially for internet access and is different from the one used by the operating system to access resources in local area networks. Chapter 4 describes the caching scheme in more detail.

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. The services provided by the Access Manager are mapped into operations of the proper access protocol.

Third, 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 2.5 schematically depicts a file that is located on a physical file system, cached on the cache server, and accessed through a workstation.

CHAPTER 3

NAME SPACE MANAGER

A Name Space Manager implements a logical name space that is the heart of the Jade file system. In addition to describing the data structures used by the Name Space Manager, this chapter gives the pathname resolution algorithm. The key data structure used by this algorithm is a skeleton directory that maintains the boundaries between the underlying file systems. A mount operation is used to construct a logical name space. The pathname resolution algorithm is more complex than those used by other file systems because a given directory in a logical name space may point to multiple underlying file systems as well as to other logical file systems. Finally, Jade supports Name Space Stacks that allow users to manage a set of logical name spaces and to perform checkpoint and rollback on mount operations.

3.1 Logical Name Space

Jade presents each user with a single tree-structure naming hierarchy. Like most Unix-like file systems, this naming hierarchy supports mapping from pathnames to file handles. However, it conceptually consists of two parts: a set of *skeleton directories* that are maintained by the logical name space, and a collection of *domains* that are supported by the individual mounted file systems. Skeleton directories glue individual domains together to form the naming hierarchy. Figure 3.1 illustrates an example of a Jade file system associated with the user John, where the dotted lines denote the partitioning of the naming hierarchy into a set of domains. For each domain, there exists a skeleton directory in the logical name space referring to the root of the domain. For example, in Figure 3.1, the domain rooted at **/jade** represents one mounted file system and there is a skeleton directory named **/jade** in the logical name space.

Skeleton directories and domains are realized in the implementation by *skeleton nodes*

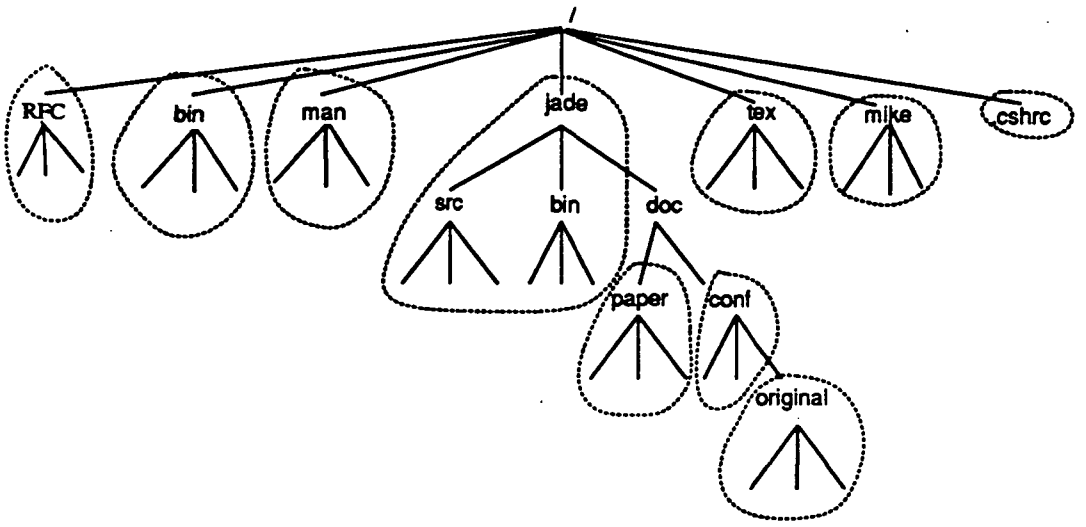


Figure 3.1: Private File Hierarchy

maintained by the Name Space Manager and physical directories as the roots of physical file systems respectively. Like mount points in the Network File System[Sun86a] and remote links in the Sprite File System[Welc86], a skeleton node serves as a pointer from a node in the logical name space to the root of a mounted file system. Unlike mount points and remote links, skeleton nodes indeed contain references to directories in mounted file systems. Entries under a skeleton directory, however, are those in the physical directory referred to by its skeleton node. The fact that skeleton directories are implemented by a Name Space Manager is transparent to the users, except that they are created by the mount operation rather than by the mkdir operation. In order to improve performance of pathname resolution, the Name Space Manager also caches directories/files inside a domain. These are called *cached nodes.*

In other words, although the user sees a Unix-like hierarchy of directories—and is able to refer to a particular directory with a pathname—the Name Space Manager implements this hierarchy as a collection of skeleton nodes and cached nodes. The skeleton nodes are intrinsic to Jade; they are similar to mount points and remote links found in other file systems. The cached nodes are caches of directory information contained in other file systems. Furthermore, even though skeleton and cached nodes are used to implement

directories in the logical name space, neither of them actually "points to" other nodes, in the same way that a directory in a Unix-like hierarchy points to some set of other directories or files. Thus, instead of locating a particular skeleton/cached node by traversing a sequence of pointers, a hash table is used to map a pathname into the skeleton or cached node that implements the directory in Jade.

Even though nodes in the Name Space Manager do not *physically* point to other nodes, in order to describe their *logical* relation, we consider a skeleton node to be a *skeleton child* of another node if the pathname of the latter is the parent-pathname of the former, where the parent-pathname of a pathname is the pathname without the last component (e.g., the parent-pathname of **/a/b/c/d/e** is **/a/b/c/d**). For example, in Figure 3.1, the skeleton node with the pathname **/jade** (called the skeleton node **/jade**) has one skeleton child, **/jade/doc**; the skeleton node "/" has seven skeleton children, **/RFC**, **/bin**, **/man**, **/jade**, **/tex**, **/mike**, and **/.cshrc**.

In this thesis, we use the term "skeleton node" to refer to the node maintained by the Name Space Manager that contains information about the mounted file systems, we use the term "skeleton directory" to refer to the directory in the logical name space that is the root of a mounted file system; it therefore implies both the skeleton node in the Name Space Manager and the physical directory in the mounted file system.

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

<Access_Protocol, Server, Handle, Token>

Access_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. Jade maintains a per-user **Tokens** list. Each **Token** in the list specifies authentication information needed to access a group of file systems and is referred to by a generic name. Section 3.6 describes **Tokens** in more detail. Until then, we consider only the first three components in the reference.

Access Protocol	*Server*	*Root handle*
UFS	Host_Name	Path_Name
NFS	Host_Name	Path_Name
AFS	Cell_Name	Volume_Name : Path_Name
FTP	Host_Name	Path_Name
JNP	User@Host_Name	Path_Name

Table 3.1: Reference

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. Table 3.1 illustrates formats for different access protocols currently supported by Jade. A logical file system can be mounted by another logical file system. The protocol used to access a logical file system, called the *Jade Naming Protocol* (JNP), is described in Section 3.5. The server that maintains the logical file system is addressed by the name of the user who owns this logical file system and the host where the server is located; it is given as **User@Host_Name**.

To illustrate better how various file systems are mounted into a logical name space, consider the following five examples from Figure 3.1. First, the skeleton node **/jade** contains the following reference:

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

In this case, **meg.cs.arizona.edu** (or **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**.

Second, the skeleton node **/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.

Third, 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@cs.arizona.edu** and is given by

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

JNP is used to access Jade name space **mike@cs.arizona.edu.**

Fourth, the skeleton node might have more than one reference. For instance, there are three references associated with the skeleton node **/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 instance, the references of the skeleton node **/RFC** are

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

Finally, the skeleton node **/jade/doc** has no reference and corresponds to no physical file systems. Notice, however, that **/jade/doc** has two skeleton children, **/jade/doc/paper** and **/jade/doc/conf.** When listing **/jade/doc**, users see two *entries* under it, i.e., **paper** and **conf.** Hence, we call the skeleton directory **/jade/doc** a ***logical directory.*** Another example of the logical directory corresponds to the skeleton node for the root "/".

3.2 Semantics of Skeleton Directories

Jade supports a mount operation that is used to attach file systems to the name space. It does this by creating a new skeleton directory in the logical name space. This section describes the mount operation and its options, and then discusses the semantics of the skeleton directories created by the mount operations with different options.

3.2.1 Mount Operation

The mount operation creates a skeleton node in the Name Space Manager with the given pathname, and associates this directory with references to mounted file systems. It is defined as

Mount(*logical_directory*, *reference_list*)

where *logical_directory* is the pathname of a skeleton node on which file systems are mounted, and *reference_list* is a list of references to mounted file systems.

In addition to the traditional mount that links a directory to one single file system, the operation supports a *null mount* and a *multiple mount* option. With the null mount option, the *reference_list* is empty. No file system is bound to the skeleton node. With the multiple mount option, on the other hand, the *reference_list* has more than one reference—the node refers to a list of mounted file systems. The list is ordered and this order is used to resolve name conflicts. Also, the mounted file system specified by the reference in the *reference_list* can be another Jade file system.

3.2.2 Reference to a Logical File System

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. As mentioned before, the reference used to refer to a logical file system is given as

<JNP, User@Host_Name, Path_Name>

For example, the skeleton directory **/mike** on John's name space (as shown in Figure 3.1) mounts the subtree **/database** of a private name space belonging to the user Mike. The reference associated with the node **/mike** is given as

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

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 3.2, Mike's name

space mounts the subtree **/bib/journal** of Bob's name space under the skeleton directory **/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.

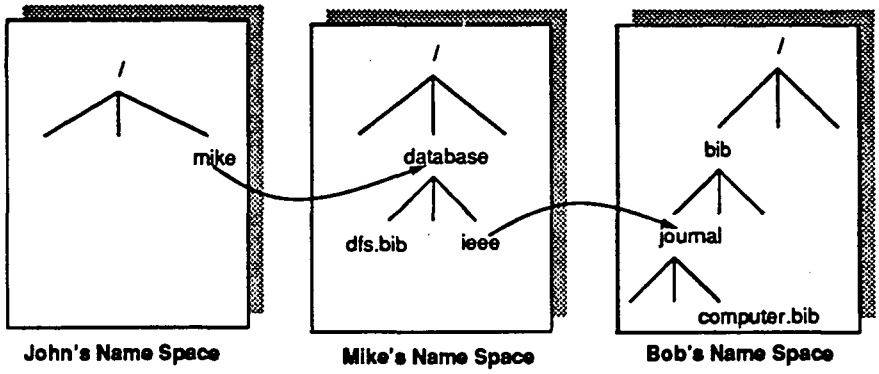


Figure 3.2: Mounting Other Name Spaces

3.2.3 Multiple Mount

The multiple mount option associates more than one file system with one skeleton directory. For example, the skeleton directory **/bin** shown in Figure 3.1 mounts three file systems: **jag:/usr/john/bin**, **meg:/usr/john/bin**, and **meg:/usr/bin**. Entries of this skeleton directory are *unioned* with those from mounted file systems. There are two new issues in regard to skeleton directories created by multiple mounts. First, entries from different mounted file systems may have name conflicts. Second, there is a question of which physical file system new files should be created on. This subsection specifies multiple mounts and discusses these two issues.

As mentioned before, entries of a skeleton directory created by multiple mounts are *unioned* with those from mounted file systems. One way to define this union operation is to *merge* entire subtrees of mounted file systems, and it is called the *union* mount. Examples of union mounts include the *viewpath* mechanism suggested by Korn and Krell's 3-D File System[Korn90] and Sun's Translucent File Service[Hend90]. In multiple mounts provided by Jade, however, the union operation is applied only to entries under the skeleton

directory; it is not recursively applied to the entire subtrees of all 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. Section 5.3 compares multiple mounts and union mounts in more detail.

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. 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. There are occasions when this semantics may be useful. For example, the software **make** may want to choose newly updated sources among several mounted file systems. We decided not to do this because it would be very hard to implement this without a internet-wide, global clock.

Finally, because a given skeleton directory refers to zero, one, or more than one file system, the file system on which a new file/directory should be created becomes an issue. Recall 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. Section 3.3 discusses how to resolve pathnames in the context of multiple

logical name spaces. 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.

3.2.4 Logical Directories and Opaque Nodes

There are two kinds of skeleton directories created by null mounts: *logical directories* and *opaque nodes*. A logical directory is a skeleton directory referring to no physical file systems, but its skeleton node has skeleton children. For example, the root (/) in the name space showed in Figure 3.1 is a logical directory with entries **RFC, bin, man, jade, tex, mike,** and **.cshrc**—each of which is a skeleton directory. An opaque node is a skeleton directory without any reference, and it is not visible under operations (e.g., listing entries of a directory) with the default option; it is shown with a special option. The opaque node is used to *hide* a file/directory with the same name from a mounted file system. Consider the name space illustrated in Figure 3.1. Users can create an opaque node with the pathname **/Jade/bin** to obscure the physical directory **/usr/john/jade/bin** and its entries located in the file system **meg**.

3.3 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 resolves 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, multiple mounts and the ability to mount logical name spaces make this procedure more complicated. 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 deals with the multiple mount. For the multiple mount, it may be necessary to try several possibilities before successfully resolving the given name.

3.3.1 Resolving Pathnames on Physical File Systems

In general, there are two approaches for resolving the remaining path on 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 uses the local pathname resolution. In order to improve performance, Jade not only maintains the skeleton nodes but also caches 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][Saty90a]. However, its functionality varies in different file systems. For example, prefix tables used by the Sprite File System map only pathname prefixes to file servers; the Network File System caches attributes and file handles of visited files/directories for later access; and the Locus Distributed System and the Andrew File System use the local pathname 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, as in Locus and Andrew, Jade starts from the skeleton directory and caches only component directories under this domain.

3.3.2 Resolving Pathnames 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

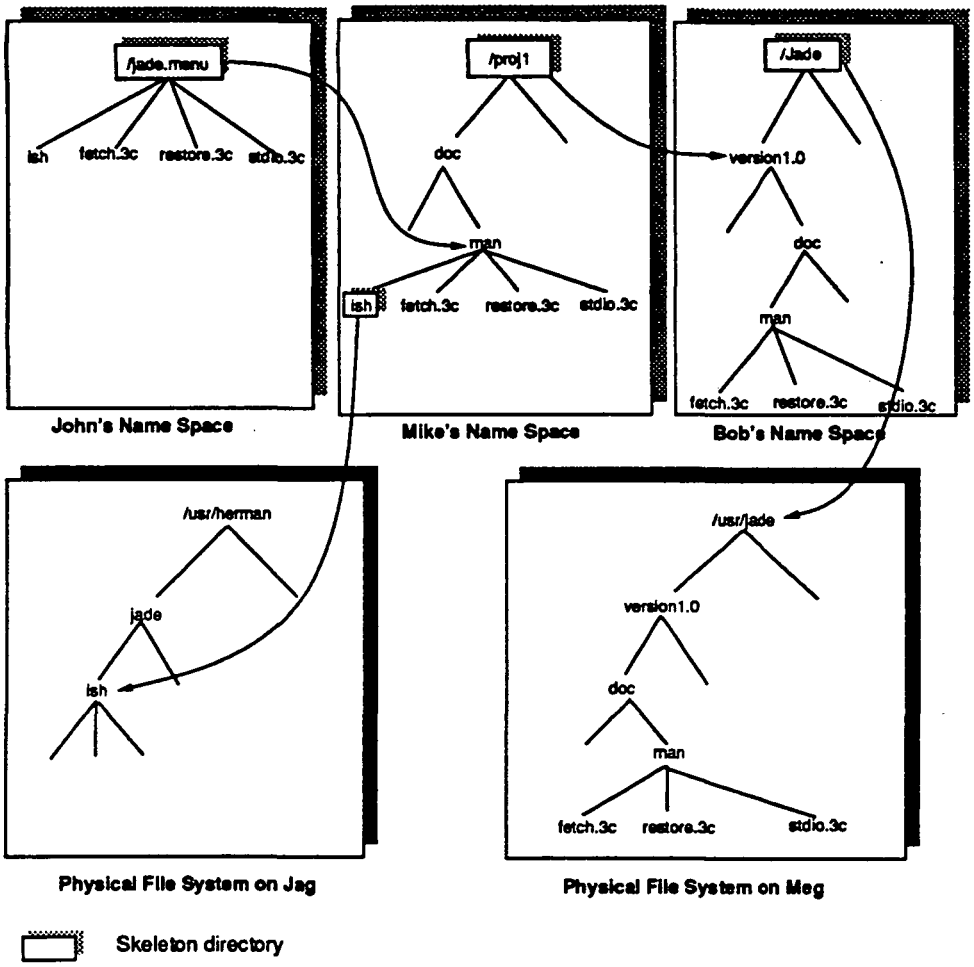


Figure 3.3: Multiple Logical Name Spaces

needs to be traversed before a desired physical file system can be located. Consider Figure 3.3, 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** refers to the same physical directory: **meg:/usr/jade/version1.0/doc/man**. In order to resolve the 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 directories on the

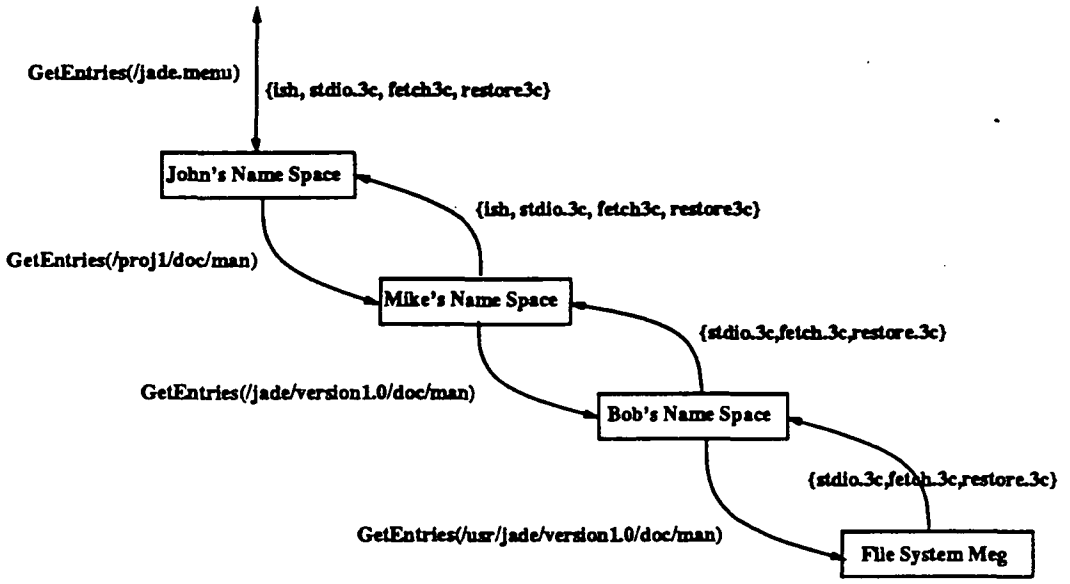


Figure 3.4: Recursive Method

name spaces of this sequence may contain other skeleton directories. For example, Mike's name space has a skeleton directory **/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 on the directory **meg:/usr/jade/version1.0/doc/man** (i.e., **fetch.3c**, **restore.3c**, and **stdio.3c**), but also **ish**. Also, name conflicts may exist between names of local skeleton directories and those from mounted name spaces. Section 3.2.3 illustrates two general rules used to resolve name conflicts. That is, names of local skeleton directories have precedence over names from mounted files systems; the order of the list of references associated with the directory is used to resolve conflicts among different mounted file systems. Finally, it is possible to form a loop within this calling sequence. How to detect and prevent loops is one of the critical issues in designing the pathname resolution algorithm. We address this issue later.

There are two methods to resolve the pathname on a sequence of name spaces: the *recursive* method and the *iterative* method. With the recursive method, pathnames are recursively resolved within the new name space, and all of the entries (including local skeleton directories and mounted files) of the directory are collected and returned to

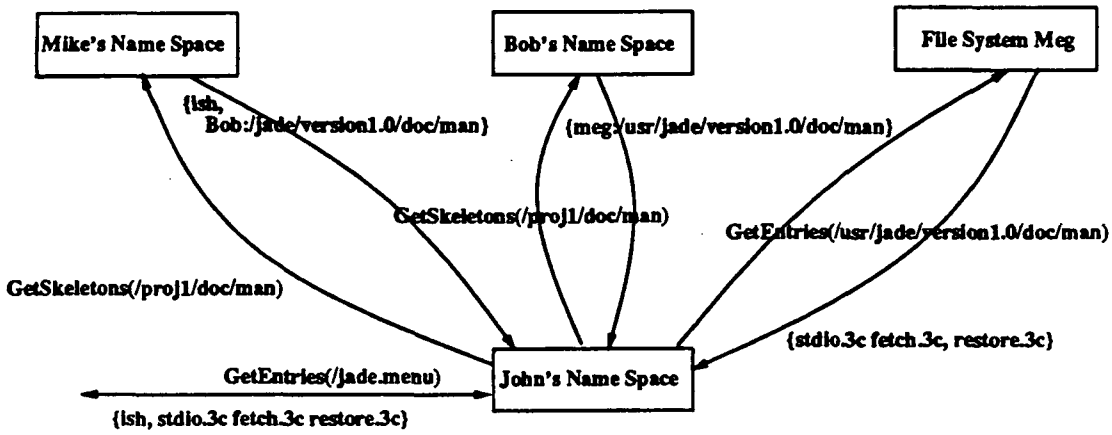


Figure 3.5: Iterative Method

the caller one at a time. Figure 3.4 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 GetEntries (See Section 3.5) 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 because 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.

Instead, we chose the iterative method illustrated in Figure 3.5. With this method, the original logical name space (i.e., John's name space) retains control over the resolution activity. When Jade calls a given name space, that name space responds to the query with the list of references associated with the queried directory and a set of skeleton nodes, each of which is a child under the queried pathname. In this 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 **ish**. John's name space then queries Bob's name space for further information, and so on. In this method, the skeleton directories are exposed, rather than hidden, by each logical name space. The GetSkeletons operation (See Section 3.5) queries the list of references associated with the node (skeleton or cached) with a given pathname and a set of skeleton children under this node. In contrast, the GetEntries operation lists all entries under one directory including skeleton nodes and regular files and directories. Because the original name space has full control over the resolution procedure, it is easy to detect loops in the mounting graph.

3.3.3 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 3.6 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 3.2.3 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**. The searching procedure terminates whenever the desired name is found in one of the name spaces of this sequence.

3.3.4 Pathname Resolution Algorithm

To summarize, the Jade file system maintains the skeleton directories and caches entries of visited remote directories; a given skeleton directory points to zero or more file systems, and each file system may be either a physical file system or another Jade file system;

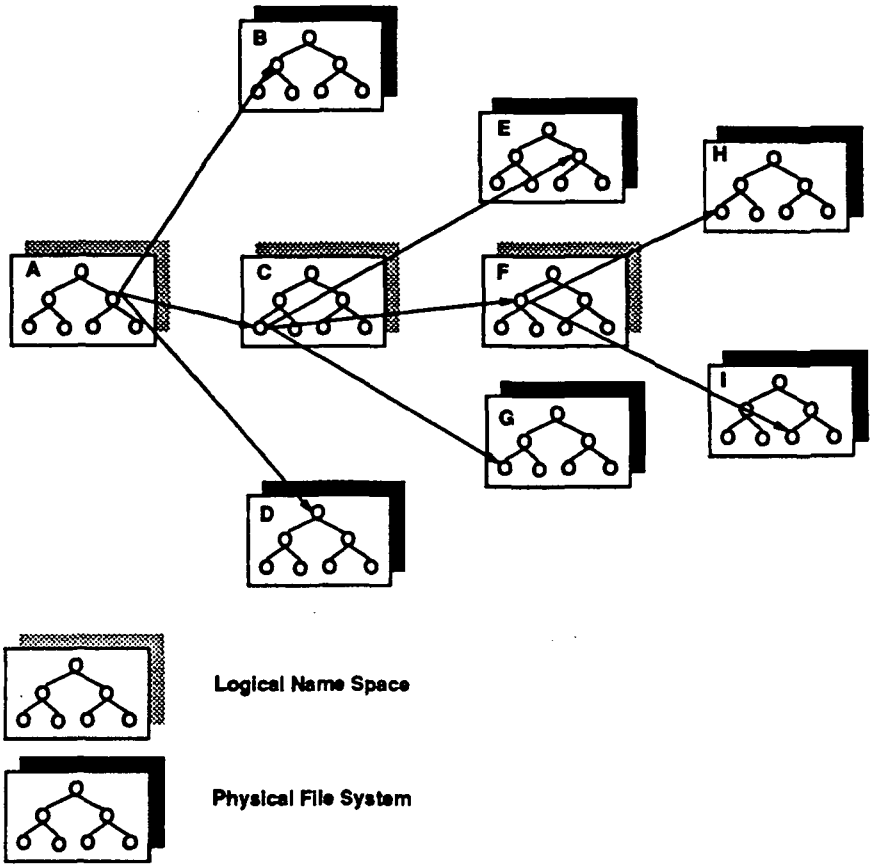


Figure 3.6: Mounting Graph

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 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 on 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()[1], translates a pathname into

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

a node. ResolvePathName() is given in Figure 3.7. It starts by locating the node in the current name space that has the longest prefix of the input pathname; it is called the *closest ancestor node.* It can be either a skeleton directory or a cached node. FindClosestAncestor() performing this search is given in Figure 3.8. 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 directory 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: An *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 detect 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 GetEntries to get entries in the remote directory and caches them. If the reference refers to a logical name space, Cache() calls the operation GetSkeletons 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 is then put in front of the *outstanding list* in order to implement the depth-first search. The inner while loop exists whenever the node with the name of the current component is located. If the *outstanding list* is empty, the function ResolvePathName() fails.

3.3.5 Listing Directory Entries

As mentioned before, skeleton nodes contain references to other name spaces, not to other nodes in the same name space, and Jade uses a hash table to locate desired nodes directly.

```
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 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);
                    /* Consults 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;
```

Figure 3.7: Function ResolvePathName()

```
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
```

Figure 3.8: Function FindClosestAncestor()

Entries under a directory include not only nodes from mounted file systems, but also skeleton directories in the logical name space. Thus, listing entries under a directory in Jade is more complicated than in other systems. Figure 3.9 outlines the Dir() function that implements this operation. In addition to the previously mentioned hash table that maps a pathname into a node, another hash table is used to implement this function. This hash table, the *Skeleton Children Table* (SCT), 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 3.1 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 node corresponding to the input pathname. Then, SCT is used to collect entries that are skeleton children under this directory node. Finally, the depth-first method presented in the previous section is used to collect other entries from the mounted file systems referred to by this directory node.

```
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
```

Figure 3.9: Function Dir()

3.4 Name Space Stack

Jade allows a logical name space to be mounted into another logical name space. This feature encourages a user to define fine grain name spaces and construct a view by overlapping a set of logical name spaces. In order to help users to organize their logical name spaces, Jade provides a Name Space Stack mechanism. The Name Space Stack also provides *checkpoint* and *rollback* on the mount operation. Jade supports push, pop, dump, and load operations to manipulate the stack.

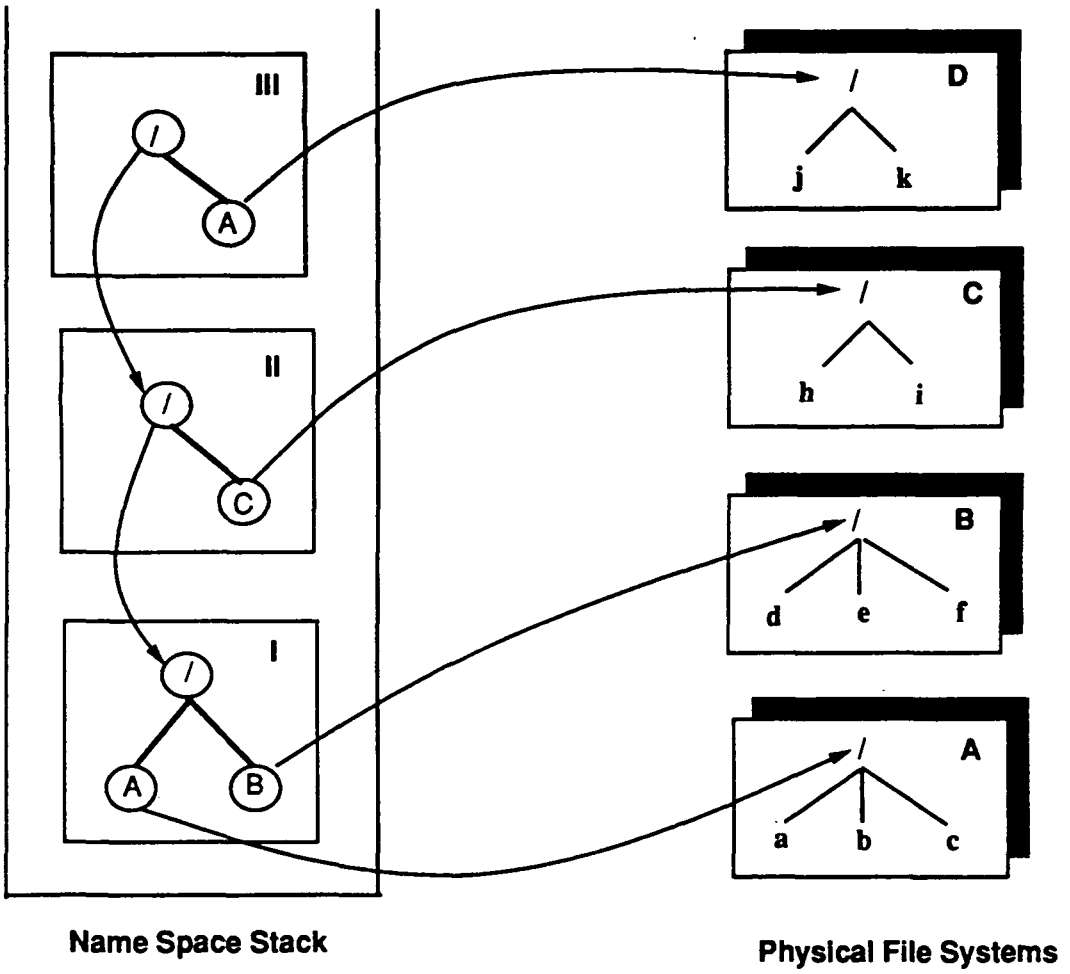
The Name Space Stack is a stack of name spaces owned by a single user. The topmost name space in the stack, called the *current working name space* (or **WNS**), is the only name space that is accessible from outside the stack. That is, all pathname resolutions are relative to **WNS**; and operations to the name space are applied to **WNS** only. For example, the mount operation creates a new skeleton directory on **WNS**, and the unmount operation removes an existing skeleton directory on **WNS**. This also means that the address of the user's name space (e.g. **john@cs.arizona.edu**) refers to his/her **WNS**. However, on the Name Space Stack, an upper name space is able to mount a lower name space. Figure 3.10 illustrates a Name Space Stack and physical file systems referred to by logical name spaces on the stack. In this example, the Name Space Stack consists of three name spaces: **I**, **II**, and **III**—**III** is **WNS**. The name space **I** mounts the physical file system **A** on the **/A** and the physical file system **B** on the **/B**. The name space **II** mounts the root of the name space **I** on its root and the physical file system **C** on the **/C**. Therefore, the view from the name space **II** includes not only the physical file system **A** and **B** but also the physical file system **C**. Finally, the name space **III** mounts the root of the name space **II** on its root and the physical file system **D** on **/A**. Consequently, from the view of the name space **III**—the current user's view—**/A** refers to the physical file system **D** rather than the physical file system **A**. More precisely, **/A** now includes files **j** and **k** which are located on the physical file system **D** rather than **a**, **b**, and **c** on the physical file system **A**.

Jade provides the operations push and pop to manipulate name spaces on the Name Space Stack. The push operation, as illustrated in Figure 3.11, generates a new space with

a single skeleton (/) pointing to the root (/) of the name space on top of the Name Space Stack. It then pushes this newly generated name space on top of the stack. After the push operation, WNS refers to this newly created name space. Note that the user's view, however, remains exactly the same. It is the mount operations following push that change the user's view. The pop operation, on the other hand, pops off the top name space from the stack. WNS changes to the next name space on the stack. Unlike the push operation, the pop operation may change the user's view. Figure 3.12 illustrates the pop operation. In this example, the pathname /A in the logical name space refers to the physical file system A instead of D after the pop operation. If the Name Space Stack only has one name space, then the pop operation fails.

In order to save and reuse name spaces, Jade supports dump and load operations. The dump operation saves an *image* of WNS to an external file. Rather than the full naming hierarchy, the image includes only skeleton nodes in the WNS. Conversely, the load operation takes an image stored on an external file, regenerates the name space, and pushes on the Name Space Stack.

There are several occasions where the Name Space Stack can be very useful. For example, a user may own several logical name spaces, each of them dedicated to a different task; e.g., one for daily administration work, one for teaching tasks, and one for research projects. With the Name Space Stack, the user is able to switch easily to different logical name spaces in order to perform different tasks, or even *overlap* more than one logical name space to have a mixed view. Version control is another possible application domain. In this case, it is possible to generate a collection of logical name spaces for a large software project, each of which represents one particular software/hardware configuration. The Name Space Stack makes it easier for the user to switch to different versions. Chapter 6 describes examples that take advantage of the Name Space Stack.



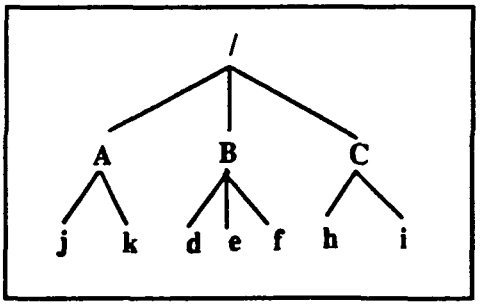
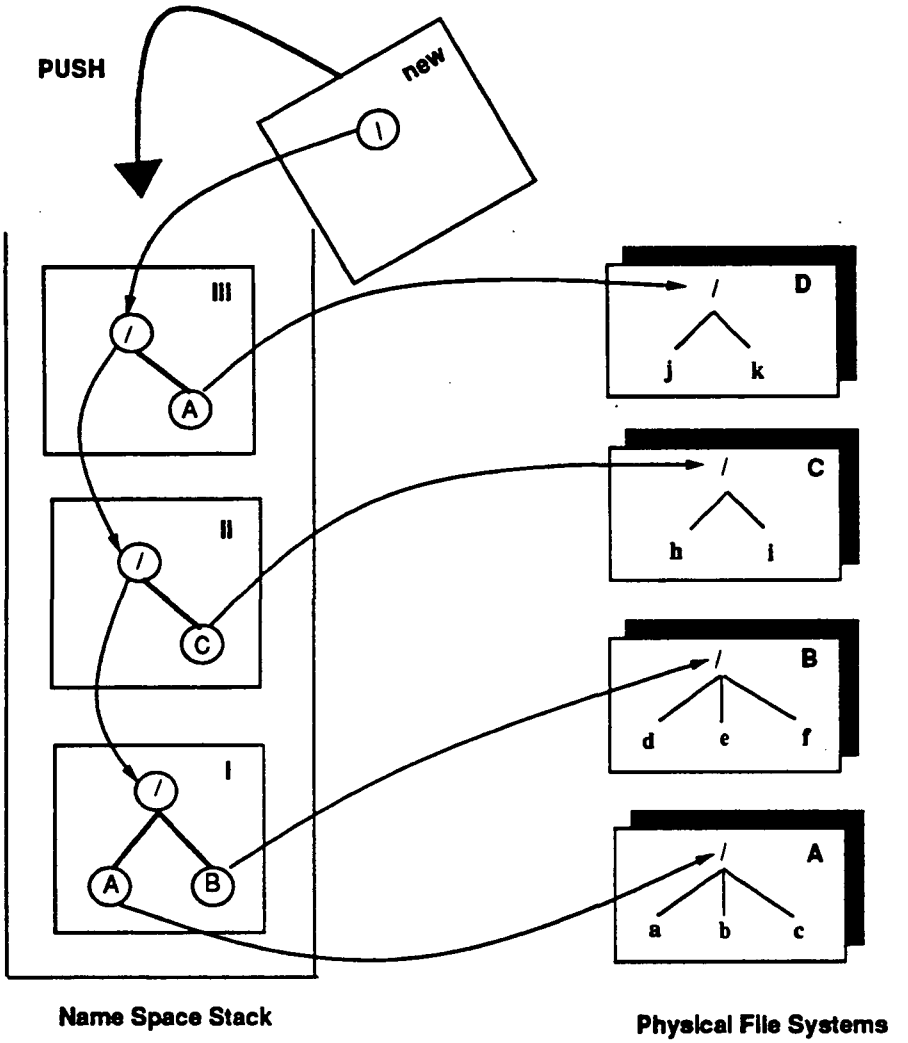


Figure 3.10: Name Space Stack



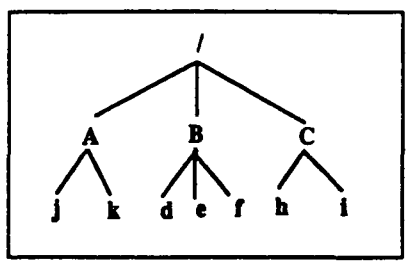


Figure 3.11: Push Operation

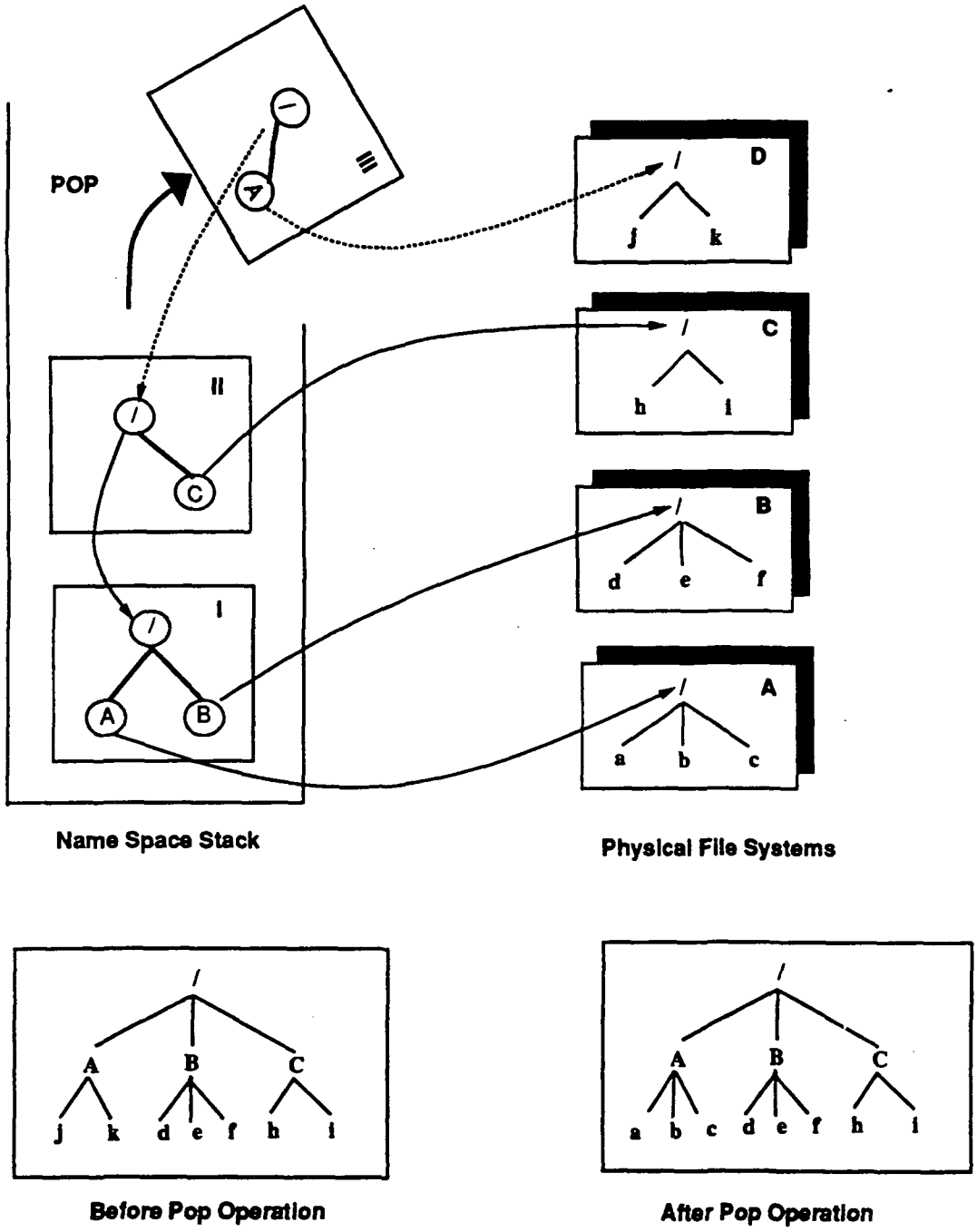


Figure 3.12: Pop Operation

3.5 Jade Naming Protocol (JNP)

The Jade Naming Protocol (JNP) specifies the interface of the Name Space Manager. A user consults the Name Space Manager with JNP in order to resolve pathnames in a logical name space. The Name Space Manager also uses JNP to consult other Name Space Managers in order to resolve a pathname. The protocol defines functions provided by a name space. These functions support directory services and can be categorized into three groups of operations: operations for handling individual files/directories, operations for manipulating a logical name space, and operations for managing a Name Space Stack. Notice that unlike the directory in other Unix-like file systems, a directory in Jade is treated differently from regular files; 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.

The protocol is specified in terms of a set of procedures, their arguments and results, and their effects. A reference to a file on a physical file system consists of four pieces of information: the server name, the access protocol supported by the server, the handle used by the server to identify the file (this handle varies by access protocol, e.g., the file handle for NFS, the inode for UFS, the fid for AFS, and the pathname for FTP), and authentication information needed to access the file in the server. Appendix A gives a complete specification of JNP. The remainder of this section discusses three interesting issues associated with JNP: listing entries of a directory, removing files, and renaming files.

Jade supports two operations to list the entries under a directory: GetEntries returns all entries (skeleton nodes and nodes on physical file systems), and GetSkeletons returns only skeleton children. Since a given directory may refer to more than one physical file system, the cost of collecting all the entries can be very high. Jade provides GetSkeletons as a less expensive alternative, and as Section 3.3 points out, this operation is useful in pathname resolution.

Jade also provides two ways to *unlink* a file: The Remove operation *physically* removes the file from the underlying file system, and the Hide operation *logically* removes the file

by creating an opaque node on the logical name space to hide it.

Finally, the Rename operation becomes complicated in the context of multiple mounted file systems. It renames the file named *path1* to *path2*, with the format:

Rename(*path*1, *path*2)

The function is successful only if the file named *path1* exists. After Rename, the file should remain on the same physical file system, and users should be able to name this file using *path2*. This implies that Rename applies only to the physical file system where the file is stored. Furthermore, if there exists a file named *path2* before invoking the Rename function, two general rules must be followed:

1. If the file *path2* is located on the same physical file system as the file *path1*, it should be removed.
2. Otherwise, the file *path2* should remain unchanged.

Consider the logical file system illustrated in Figure 3.13, where /A mounts the physical file system A, /B mounts the physical file system B, /AB mounts A and B, and /BA mounts B and A. Notice that the mounting sequence is significant and, therefore, /AB/a refers to the file A:/a, while /BA/a refers to the file B:/a.

For a domain with a single reference, Rename behaves as in Unix. For example,

Rename(/A/a, /A/b)

returns success and the original file named A:/b is removed and A:/a is renamed as A:/b. However,

Rename(/A/b, /B/i)

fails. This is because /A/b and /B/i refer to different physical file systems. For domains with multiple references, the situation becomes more complicated. For example,

Rename(/AB/c, /AB/f)

is successful—the file named A:/c is renamed as A:/f. Notice that the file B:/f is unchanged, and therefore the path /B/f and the path /AB/f, which used to point to the same file, now refer to different files. As another example,

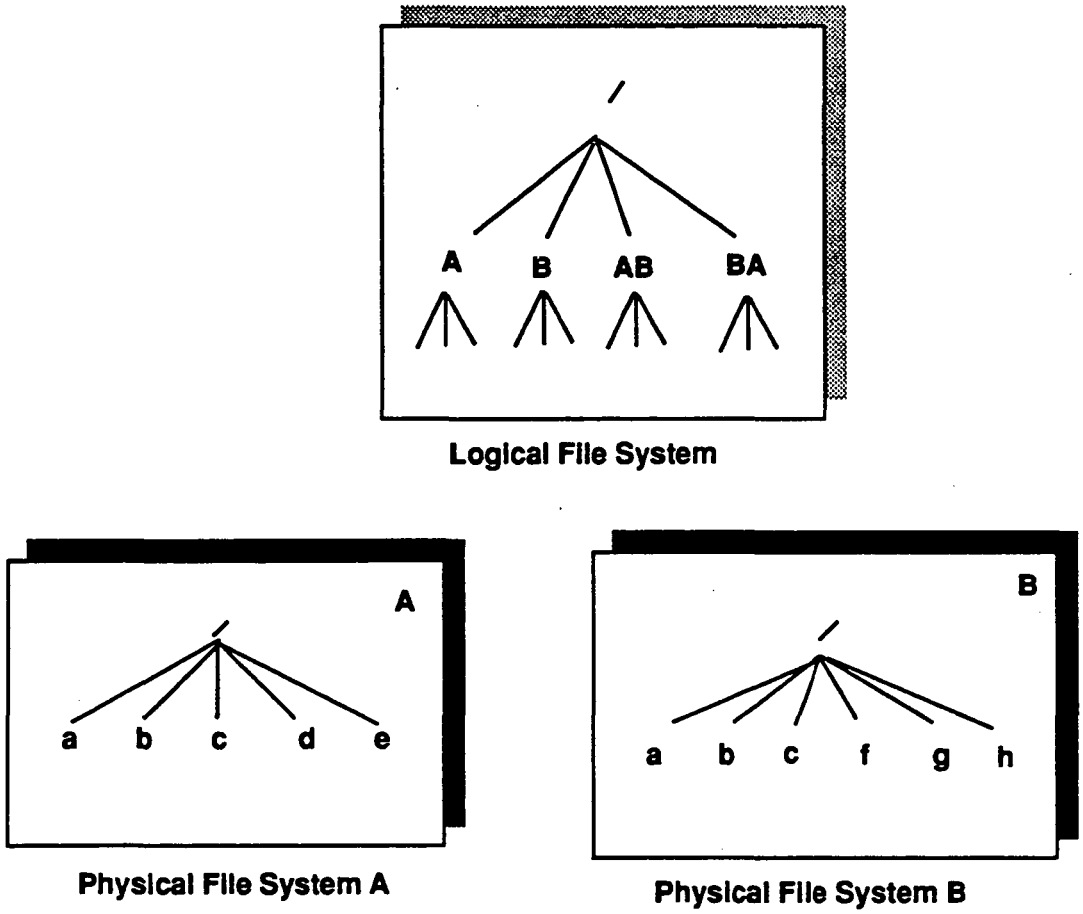


Figure 3.13: Renaming Files

Rename(/AB/g, /AB/i)

succeeds. However, the file is still located on B and the path /AB/i refers to the file B:/i rather than A:/i. As a final example,

Rename(/AB/h, /AB/d)

fails. This is because even if it were successful, the path /AB/d still refers to the file A:/d rather than to the file B:/d which was renamed from B:/h.

3.6 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 user and the file system; it does not have any special privileges. Also, the end-to-end augment[Salt84] suggests that functions placed at intermediate levels of a system may be redundant or of little value when compared with the cost of providing them at intermediate levels. Therefore, we believe that the authentication mechanism should be installed on the server where objects are implemented, rather than on the intermediate agent. 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.

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. A **Token** list is maintained on a per-user basis. 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**. Recall that each reference specifies a name of the token that is used to access the file system. For example, a reference associated with the pathname **/RFC** in the name space illustrated in Figure 3.1 is

<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 (i.e., FTP in this example).

Unix access control requires that a user have access right 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 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 prefix 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.

CHAPTER 4

ACCESS MANAGER

This chapter focuses on Jade's second major component: the Access Manager. The Access Manager supports access to a remote file given by its reference. It allows users to choose one of several physical file systems as the cache server, and caches the entire file on this server. In order to promote maximal sharing of these cached copies, the Access Manager records files that are accessed or have been accessed by clients. It supports operations to request and release a cached copy. Jade uses a two-level cache. The operating system supports the first-level cache: caching files in memory when access files from the cache server. The Access Manager caches remote files on the cache server, which is considered as the secondary cache. In order to reduce the number of messages sent to the underlying file systems, the Access Manager implements two delayed-write policies: *write-on-close* and *create-on-close.*

4.1 System Structure

The Access Manager records files that are accessed or have been accessed in a table called the *jnode* table. Each entry in the table, called a *jnode*, represents a file that is maintained and cached on the cache server. It consists of a source reference, a sink reference, a cache reference, a timestamp, and a reference count. Figure 4.1 illustrates the structure of a jnode. Section 3.1 defines the contents of a reference. It includes the name of the protocol used to access the mounted file system, the name of the host that provides services to access the mounted file system, the handle used by the server to identify the root of the mounted file system, and authentication information used to access the mounted file system. The source reference refers to a file on a physical file system from which the cached copy is fetched. The sink reference, on the other hand, points to a file on a physical file system where the cached copy is supposed to be placed when it is restored back. By default,

the source reference and the sink reference refer to the same file on the same physical file system. Jade provides an operation to change the sink reference. This is useful when users want to copy files from one physical file system to another; Section 4.2 describes this use in more detail. The cache reference refers to the cached copy on the cache sever. The timestamp marks the timestamp of the source when the file was fetched. The reference count records the number of clients that are accessing the cached copy of this file; it is used to implement a replacement mechanism. The Access Manager assigns an unique number to each jnode, called the *jnode number*.

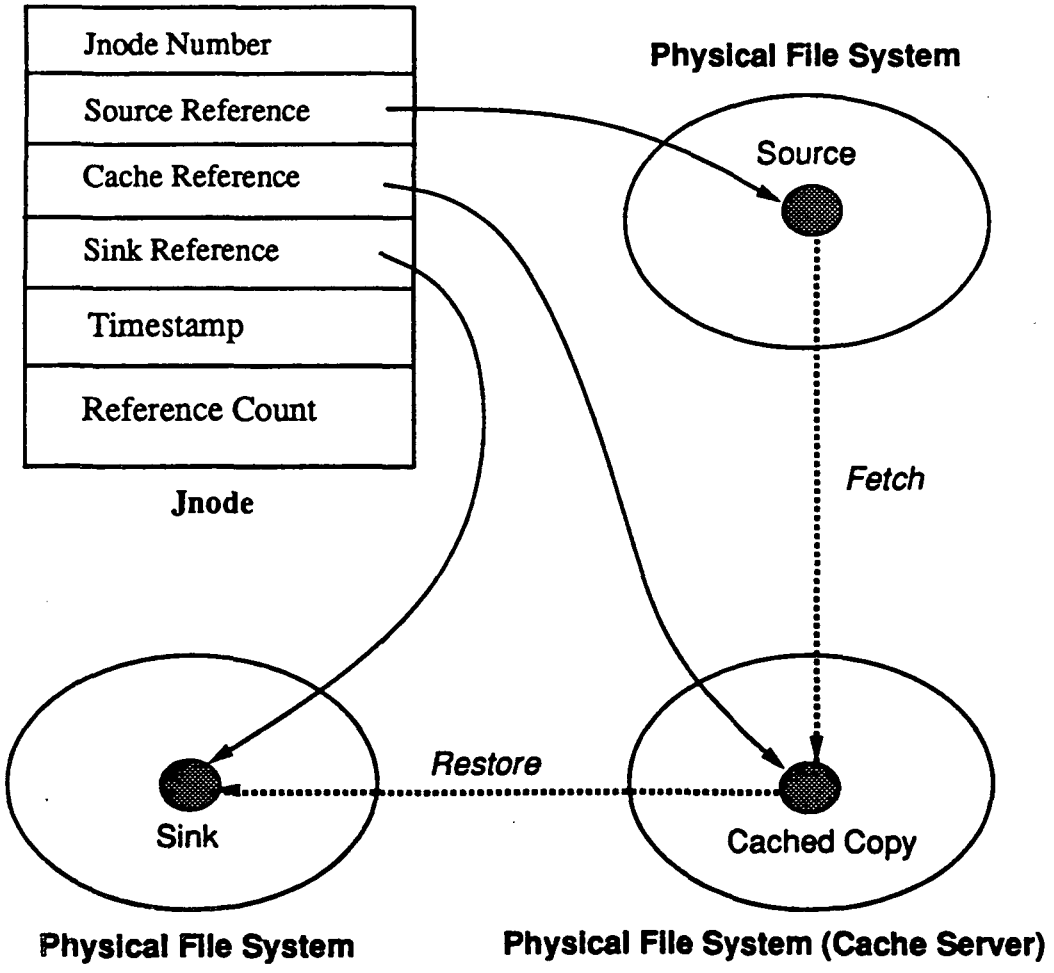


Figure 4.1: Jnode Structure

The Access Manager provides the following three operations: Request, Release, and Relabel. The Request operation locates a jnode in the jnode table, validates the cached copy, increments its reference count, and returns the cache reference specified in this jnode. The Release operation decrements the jnode's reference count and restores the cached copy of the file back to its sink, if necessary. The Relabel operation changes the sink reference of a jnode.

More precisely, the Request operation is defined as

$$cache\_reference, jnode\_number := \text{Request}(source\_reference, timestamp, flag)$$

where *source_reference* is used as a key to locate the jnode from the jnode table. If the desired jnode is not found, the Access Manager allocates a new jnode structure, initiates its source reference and sink reference to *source_reference*, and requests a unique pathname from the cache server that is assigned as the jnode's cache reference. The *timestamp* argument is used to validate the cached copy referred to by the cache reference. This validation procedure is defined as follows. The *timestamp* is compared with the one stored in the jnode structure. If this comparison indicates that the cached copy is invalid, the Access Manager fetches—using the Fetch operation of the underlying access protocol—a fresh copy from the source, and replaces the cached copy pointed to by the cache reference. Notice that the Access Manager does not implement its own clock; it only records and compares the timestamp associated with a file in the source file system.

The *flag* argument indicates the *type* of request:

- *request_for_read*;
- *request_for_write*;
- *request_for_read_and_write*.

With the flags *request_for_read* and *request_for_read_and_write*, the Access Manager validates the cached copy and fetches a new copy if necessary. With the flag *request_for_write*, on the other hand, the Access Manager ignores the validation procedure and simply returns the cache reference. It delays the creation of new files until they are closed; this is called *create-on-close*. The next section addresses this issue in more detail.

The Release operation releases a previous request. It is given as

status := Release(*jnode_number*, *flag*, *mode*)

where *jnode_number* is used to directly locate the proper jnode in the jnode table. In addition to decrementing its reference count, the Release operation restores—using the Restore operation of the underlying access protocol—the cached copy back to the sink reference. The *flag* argument indicates the restoring process as being either:

none: no restore is done and the cached copy is just freed;
synchronous_write: restores back before the Release operation returns;
asynchronous_write: schedules the restoring process and returns without waiting for its completion.

Thus, Jade is capable of supporting two kinds of writing policy: synchronous-write and asynchronous-write. The latter is especially important in performance because the remote file system may be located anywhere in the internet and may even be temporarily inaccessible. The target file is created with the mode *mode*.

Finally the Relabel operation is defined as

status := Relabel(*jnode_number*, *sink_reference*)

The sink reference of the jnode identified by *jnode_number* is changed to *sink_reference*. The Relabel operation is used to avoid unnecessary duplications of cached copies. The next section addresses this issue in more detail.

The result of this design can be best understood by examining a file access in detail. Suppose a user process opens a file with pathname P. The Name Space Manager is consulted to resolve P. It returns a source reference S to a file on some physical file system, and the timestamp T associated with this file. The reference includes the name of the access protocol used to access the underlying file system, the name of the server that supports access to the file system, the file handle that is used by the server to identify the desired file, and authentication information needed to access the file system. The Request operation is then invoked with arguments S and T to get the reference to the cached copy on the cache server. According to the way in which the file is opened (i.e., open for read,

open for write, or open for read and write), the *flag* argument is set (i.e., *request_for_read*, *request_for_write*, or *request_for_read_and_write*). The cached copy is then opened, and subsequent read and write operations are directed to this cached copy. When the user closes the file, the cached copy is closed first. Finally, the **Release** operation is called; its *flag* argument is set according to the needs of the restore process.

4.2 Caching Scheme

Jade supports two-level caches: Remote files are cached on the cache server by the Access Manager, and files on the cache server are cached on the client workstation by the operating system. With this scheme, the cache server is considered to be the secondary cache. There are two advantages of this approach. First, it allows the access protocol used to fetch the remote file to be different from the one used to access the cached copy. This approach suggests that resources other than files—e.g., mailboxes and printers—can also be named and accessed through Jade. When *fetching* a remote resource, the Access Manager generates a local, file-like *object* of the resource on the cache server. This object is then transferred back to the proper form when it is restored.

Second, experiments with the prototype of Jade show that performance factors change when files are located in an internet rather than in a local area network. For example, network latency, which is insignificant in comparison with local computations, becomes significant in the internet environment. The two-level cache scheme permits a different cache policy for each environment. In order to get good performance, it is common to choose the default file system used by the workstation as the cache server. That is, the local disk is chosen as the cache server for diskful workstations. Otherwise, a nearby file server is selected. The operating system in a client workstation uses page access rather than file access to access files in the cache server. However, like the Andrew file system, the cache server caches entire files on the disk, which is different from the memory cache scheme adopted by most other distributed file systems. Caching entire files is essential in access files in the internet. This is because the network latency in the internet becomes higher and communication is not as reliable as in the local area network. Caching entire files can avoid individual block requests as well as transfers that may be very expensive or

even fail in the case of a temporary network partition. Chapter 5 addresses performance issues in more detail.

One drawback of this caching scheme is that the cost of copying files is more expensive than the memory cache scheme. Figure 4.2 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. In 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 that 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. In an optimized case, Jade's Relabel operation lets users change the sink reference associated with the cached copy to a new one, 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 ideal way to solve this problem is to have the access protocol support a new *copy* operation, thereby making the cost of copying files comparable to the cost of *renaming* files.

Jade's *write-on-close* policy means that dirty files are written back to the underlying file system only when the files are closed. This delayed-write scheme has two advantages over a simple write-through scheme. First, because writes are to the cache, write accesses complete much more quickly. This is particularly true in Jade because individual write operations to the file system need to go through the internet, and this is very expensive. Second, data may be deleted before they are written back, in which case they do not need to be restored at all. However, this policy requires the closing process to delay while the file is written through. In order to overcome this problem, Jade provides a *write_behind* policy as one option of the Release operation. With this option, the close operation returns before data written back to the physical file system. This is particularly important because the target file system may be located anywhere in the internet and may even be temporarily unreachable due to network partition.

In addition to this *write-on-close* policy, Jade supports a *create-on-close* policy that creates a new file on the target file system when the file is closed. Most distributed file systems need a file handle for the new file before data can be written on the cache, and this file handle is issued by the target file system. In Jade, a cache reference is generated by

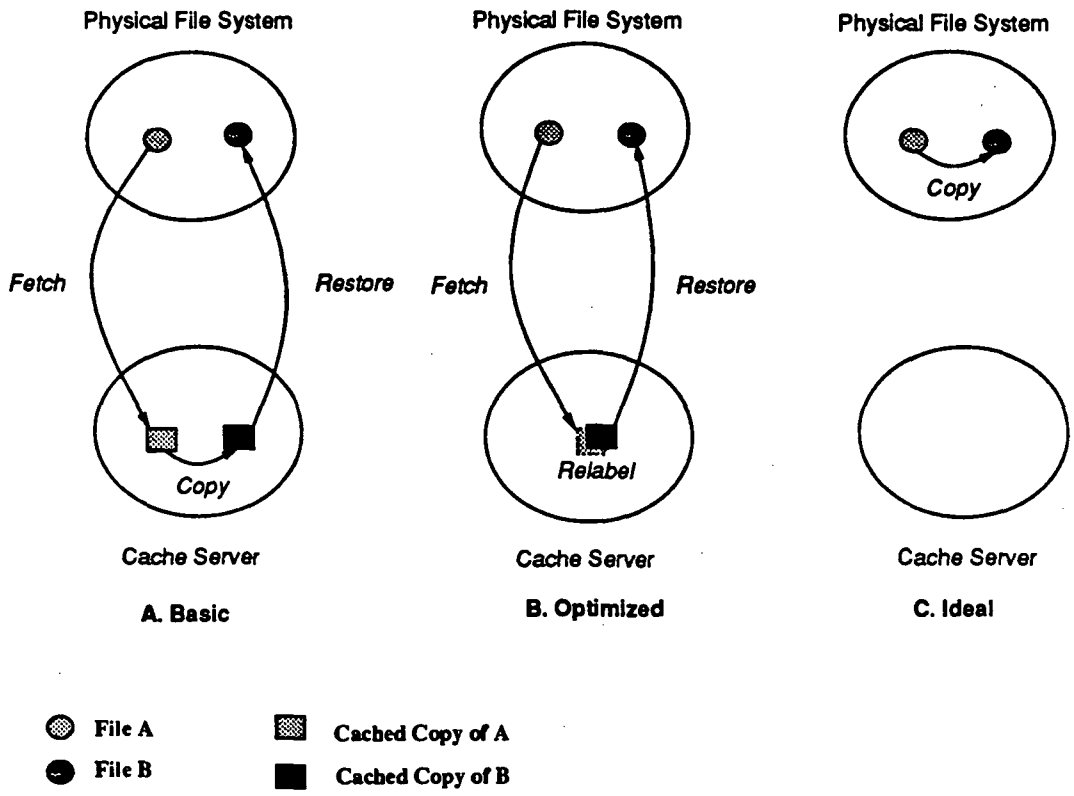


Figure 4.2: Copying Files

the Access Manager without consulting the target file system, and it is used to write data on the cache. The file system is not aware of the existence of this new file until it is closed and stored to the file system. The reason behind this design is that experiments have shown most file lifetimes are very short[Shel86][Howa88][Oust85], where these lifetimes are measured starting from the creation of the file on the cache. A trace-driven analysis on the Unix file system[Oust85] shows that 50-60 percent of such files have a lifetime of less than 3 minutes. A further observation finds that most of these files are temporary files used to pass data between sequence executions; they are deleted right after the next execution is finished. For example, in program development, the compiler generates an assembler file which is deleted as soon as it has been translated to a machine code. The *create-on-close* policy delays the act of creating a file on the underlying file system until the file is closed on the cache server.

CHAPTER 5

EVALUATION

In order to examine the design, we have implemented a prototype of the Jade File System and measured its performance. This chapter reports the experience. The prototype consists of interfaces to the access protocols UFS, NFS, and FTP. We measured the performance of this prototype with the Andrew Benchmark[Howa88]. The last section in this chapter re-examines major issues in the design and implementation of the Jade file system, with an emphasis on the tradeoffs of alternative choices.

5.1 Prototype

The prototype of Jade is implemented on top of the Sun OS 4.1. operating system and located at the user-level without any modification to the kernel. It uses Sun RPC[Sun86b] as the interprocess communication mechanism between system components. Jade adopts the Sun Shared Library mechanism[Sun88]. By dynamically linking Jade's shared library, most existing software (*ed*, *cc*, *find*, etc.,) is able to transparently access Jade without modification or re-compilation. The prototype of the Jade file system co-exists with the file system supported by the operating system of the workstation (called the *original file system*) in that Jade is *attached* on top of the directory **/Jade** of the original file system. Figure 5.1 illustrates one example of name spaces for a workstation user, including a per-user based name space provided by the Jade file system and a shared name space supported by the original file system.

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. It is easy to experiment and to examine 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. Portability among heterogeneous operating

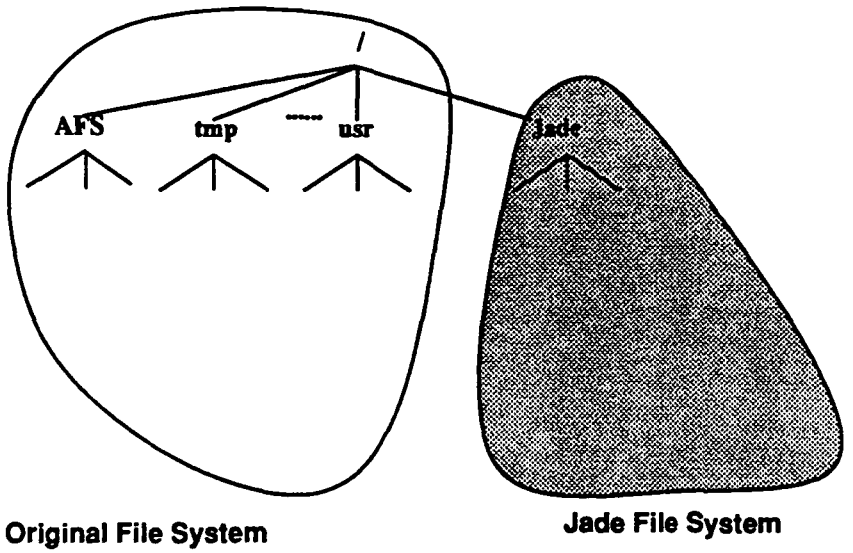


Figure 5.1: Name Spaces for a Workstation User

systems is another advantage. 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.

The prototype includes two user-level servers, one shared library, and a collection of agents for different access protocols, as illustrated in Figure 5.2. The first server implements the Name Space Manager described in Chapter 3, while the second server implements the Access Manager explained in Chapter 4. Jade implements the Name Space Manager and the Access Manager as separate servers rather than combining them. This is because the Name Space Manager is defined on a per-user basis, whereas the Access Manager allows different users to share a single cache. The shared library embeds functions into the system call interface. These functions invoke services from the Name Space Manager for pathname resolution and services from the Access Manager for file caching. Sun RPC is used as the communication mechanism between the shared library and the two servers. Jade implements each of the access protocols (i.e., UFS, NFS, and FTP) as an *agent*. Agents support a uniform interface on which the Name Space Manager and the Access Manager are implemented.

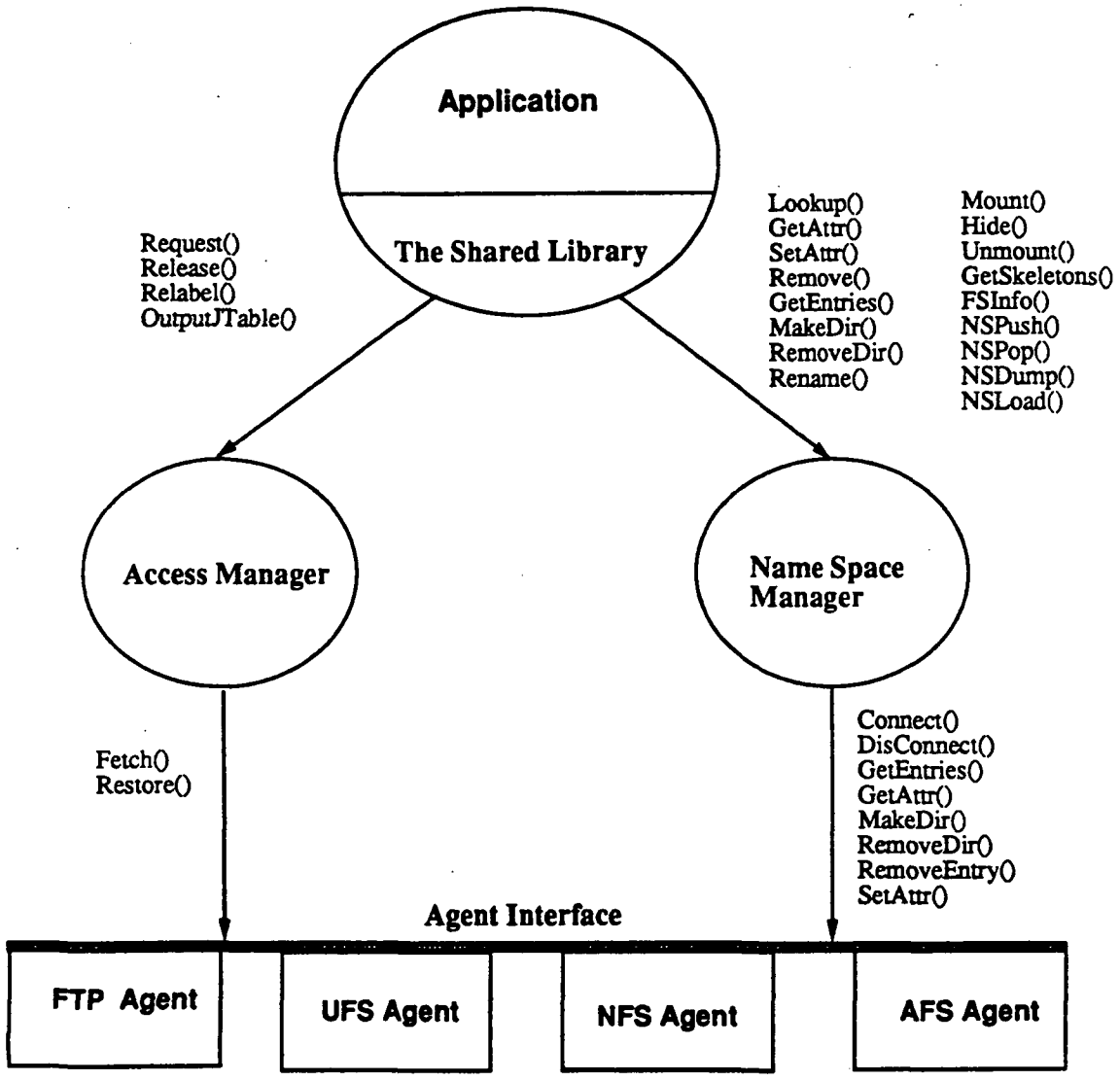


Figure 5.2: Implementation Structure

```
open(path, flags, mode)
    reference := NameSpaceManager.Lookup(path);
    if reference is not nil then
        handle := AccessManager.Request(reference, request_for_read);
        if handle is not nil then
            return syscall(SYS_open, handle, flags, mode);
        fi
    fi

    return fail;
```

Figure 5.3: Function open

In order to present how the shared library and two servers are tied together, consider the function open in the shared library as illustrated in Figure 5.3. To simplify the discussion, only the function of opening a file for reading is described. When a file is opened for read, the Name Space Manager is consulted in order to resolve the given *path*. It returns a reference to a file. The Access Manager is then invoked to cache the entire file using the reference as an argument. A handle of the cached copy is then returned. The cached copy is located in the original file system, and the handle is either a pathname or an *inode* defined in the file system. Finally, the cached copy is opened by invoking the system call syscall.

The remainder of this section discusses the implementation of each of four components. We start from the bottom with protocol agents. Next, we describe the Name Space Manager and the Access Manager, which are implemented on top of protocol agents. Finally, we depict the shared library, which invokes services of the Name Space Manager and the Access Manager.

5.1.1 Protocol Agents

In order to accommodate heterogeneous access protocols, the Access Manager and the Name Space Manager define a uniform interface through which services supported by these two managers are mapped into functions of individual protocols. More precisely, for each access protocol, there is an agent that supports the uniform interface for the protocol. Each agent implements the client part of a protocol and encapsulates communication details of this protocol (e.g., Sun RPC[Sun86b] for NFS, Rx[Side89] for AFS, and TCP[Post81] for FTP). An agent serves as the front-end to a collection of file systems accessible through the protocol.

Each agent supports a set of functions, as summarized in Table 5.1. The function Connect starts the dialogue with the physical file system and returns the handle of the connection. The function Disconnect terminates this connection.

In order to implement the local resolution method described in Chapter 3, the function GetEntries retrieves entries under a given directory of the physical file system. Notice that there is no lookup function defined in agents. This is because pathname resolution is done in the Name Space Manager rather than in physical file systems.

The function Fetch (Restore) retrieves (stores) a file from (to) the physical file system. Because Jade supports entire file caching, agents do not support Read and Write functions to access individual pages as the Network File System does. Because of the create-on-close semantics, new files are created by the function Restore and there is no creat function.

Other functions defined by agents support common directory services. The function GetAttr (SetAttr) retrieves (sets) attributes associated with a file/directory in a file system. The function RemoveEntry removes an entry under a directory in a physical file system. The function MakeDir creates a new directory on a physical file system. The function RemoveDir removes an existing empty directory on a physical file system.

The following reports our experience implementing agents for UFS, NFS, and FTP. In addition to describing the prototype, we report problems in implementing the agent for each protocol.

Connect	connects the server that supports a physical file system and returns a handle for the file system.
Disconnect	disconnects the server.
GetEntries	gets entries under a directory in a file system.
RemoveEntry	removes an entry under a directory in a physical file system.
GetAttr	returns attributes associated with a file or directory in a file system.
SetAttr	sets attributes of a file/directory in a physical file system.
MakeDir	creates a new directory on a physical file system.
RemoveDir	removes an existing, empty directory on a physical file system.
Fetch	retrieves an entire file from a physical file system.
Restore	stores data back to a file in a physical file system.

Table 5.1: Agent Interface

UFS Agent

The implementation of the UFS Agent is trivial. Most of the functions are directly mapped into the corresponding system calls in the Unix file system interface[Bach86]. For example, the function MakeDir is mapped into the system call mkdir; the function GetAttr is implemented by the system call stat. Because the desired file system is located on the local host, there is no need to connect the file system at the beginning, and therefore, the functions Connect and Disconnect invoke no Unix system calls.

NFS Agent

The NFS Agent communicates with file servers using Sun RPC. Like the UFS Agent, most of the NFS Agent's functions are mapped into corresponding RPC calls directly. For example, the function GetEntries is implemented by the RPC call NFSPROC_READDIR; the function RemoveEntry is mapped into the RPC call NFSPROC_REMOVE. However, the function Connect invokes the call MOUNTPROC_MNT, which is supported by the Sun's Mount Protocol[Sun86a] instead of the NFS protocol, to get the handle of the root of the mounted file system. Furthermore, NFS supports page access instead of file access as used in Jade. Therefore, the function Fetch (Restore) opens a file and then invokes sequences

of the call NFSPROC_READ (NFSPROC_WRITE) to retrieve (store) the entire file.

The major drawback of the NFS protocol is that it was designed based on the existence of a unique identifier (uid) for each user. More precisely, clients identify themselves to the server with their uids rather than login names; the server returns owners' uids for file attributes instead of string names, and the protocol supports no functions to convert between a uid and its login name. In order to handle this problem, Jade keeps a uid in addition to a login name in the corresponding **Token** (see Section 3.6).

FTP Agent

The FTP Agent is implemented on top of the Unix socket interface. It communicates with file servers through Transmission Control Protocol (TCP)[Post81]. The agent's functions are mapped into FTP commands. The function Connect initiates dialogue with a server. Transferring files is straightforward in the FTP Agent: The function Fetch invokes the command RETR to retrieve the whole file from the file server, while the function Restore calls the command STOR to store a file into the file server.

Unlike access protocols for other agents, the FTP protocol specification does not support all the functionality needed to implement an agent. There are two major problems. First, the content and format of attributes of files/directories are undefined in the protocol specification, and therefore, they may vary from one file server to another. The function GetAttr invokes the command LIST to get attributes of a file/directory from the server. However, there is no common way to parse the returned attributes. Furthermore, FTP does not support commands to change attributes associated with a file/directory and, therefore, the function SetAttr is undefined in the FTP Agent. Second, the notion of directory is defined as an option in that it is not supported by every file server. Therefore, the function MakeDir and RemoveDir, which are implemented by the FTP command MKD and RMD, are not available for all file systems.

5.1.2 Name Space Manager

The Name Space Manager is implemented as a server. Each machine has a single daemon that is able to support a collection of logical name spaces at one time. The name space

is addressed as **UserName@HostName** where **HostName** specifies the host on which the server resides and **UserName** is an identifier used by the daemon to specify a logical name space. The name space, consulted by a user process, is addressed by the environment variable **NameHost**.

Functions supported by the Name Space Manager are categorized into three groups: functions for regular directory services (i.e., Lookup, GetAttr, SetAttr, Remove, GetEntries, MakeDir, RemoveDir, and Rename), functions for maintaining a logical name space (i.e., Mount, Unmount, Hide, GetSkeletons, and FSInfo), and functions for handling a Name Space Stack (i.e., NSPush, NSPop, NSDump, and NSLoad). The following describes each of these functions; Table 5.2 summarizes them.

The function Lookup takes a pathname as an argument and returns a *reference* to a file. The reference includes the name of the access protocol used to access the physical file system, the name of the server maintaining the physical file system, the handle used by the server to identify the file, and the authentication information needed to access the server (see Section 3.1).

The functions SetAttr and GetAttr are used to manipulate attributes associated with a file/directory. The function Remove deletes a file with a given pathname. The function Rename changes the name of a file or a directory (the semantics of Rename is discussed in Section 3.5). The function MakeDir creates a directory in the physical file system. The function RemoveDir removes an existing empty directory on a physical file system.

The function Mount creates a new skeleton directory with the given pathname and associates it with a list of references. The function Unmount deletes an existing skeleton directory. The function Hide is a special case of the function Mount; it creates an opaque node—a skeleton node without any reference as described in Chapter 3. Jade supports two different functions to make a directory entry invisible: the function Remove, which unlinks a file in the physical file system, and the function Hide, which hides the named file by creating a opaque node.

The function GetSkeletons lists skeleton directories under the specified pathname. Each entry contains a pathname, a list of references, and attributes. This function is used to implement the iterative search on a sequence of name spaces. The function FSInfo returns

the skeleton directory of the domain pointed to by a given pathname.

Finally, the Name Space Manager supports functions to manipulate the Name Space Stack as described in Chapter 3. The function NSPush pushes a new name space on top of the Name Space Stack, while the function NSPop pops off the top name space of the Name Space Stack. The latter returns error if the Name Space Stack becomes empty after the pop operation. The function NSDump outputs the top name space of the Name Space Stack to a file, while the function NSLoad generates a new name space from a file and pushes it on top of the Name Space Stack.

5.1.3 Access Manager

The Access Manager maintains the jnode table and caches remote files on the cache server. The location of the cache server is specified by the environment variable **CacheHost**. By assigning a new address to this variable, users are able to move the Access Manager from one host to another. There are a number of situations in which such a migration is desirable. One of the most common cases concerns accessing large files or even databases. The physical file system currently used as a cache server may not have enough room to cache remote files. Users can then migrate the Access Manager to a host with a larger file system, or even to the host where the desired file is located. In the latter case, the caching process is avoided.

The Access Manager supports four functions: Request, Release, Relabel, and OutputJnodeTable; Table 5.3 summarizes these functions.

The function Request takes a reference to a file as an argument and returns a handle of the cached copy of the remote file on the cache server. The handle is either a pathname or an *inode* of the cached copy in the cache server and is accessible through the original file system. The function supports three different kind of access: *request_for_read*, *request_for_write*, and *request_for_read_and_write*.

The function Release dismisses a cached copy by decrementing its reference count. It restores the file back to the sink file system if necessary. The restore process is defined by a input flag that is either *none*, *write_through*, or *write_behind*.

Finally, the function Relabel changes the sink reference of a given jnode, while the

Lookup	returns a reference to the named file.
GetAttr	gets the attributes and the file reference of a file/directory.
SetAttr	sets the attributes of a file/directory.
Remove	deletes a specified file.
GetEntries	lists entries under one directory.
MakeDir	creates a new directory on a physical file system.
RemoveDir	removes an existing empty directory on a physical file system.
Rename	changes the name of a file or a directory. Cross file system renames are illegal.
Mount	creates a new skeleton directory and mounts the specified file systems on this directory.
Hide	creates an opaque node with the given pathname.
Unmount	removes the specified skeleton directory and unmounts file systems on this directory.
GetSkeletons	lists skeleton directories under the specified directory. Each skeleton contains a pathname, a list of references, and attributes.
FSInfo	returns the skeleton directory of the domain in which the given pathname is located. The skeleton consists of a pathname, a list of references, and attributes.
NSPush	creates a new name space and pushes on top of the Name Space Stack.
NSPop	pops off the top name space of the Name Space Stack.
NSDump	outputs WNS to a file.
NSLoad	creates a new name space from the specified file and pushes on top of the Name Space Stack.

Table 5.2: Name Space Manager Interface

function OutputJnodeTable outputs the jnode table to a file.

Request	requests a cached copy for a remote file and returns the handle of the cached copy.
Relabel	changes the sink reference of the given jnode.
Release	releases a cached copy and restores the file if necessary.
OutputJnodeTable	outputs the jnode table to an external file.

Table 5.3: Access Manager Interface

5.1.4 Shared Library

Jade modifies system calls in order to provide transparent access to the Jade file system. As mentioned before, the prototype of the Jade file system is *mounted* on top of the directory **/Jade** in the original file system. That is, file names with the prefix **/Jade/** are handled as names in the Jade name space; other file names are handled as names in the original file system. This mounting process is different from the mount operation provided by the original file system or by Jade. Instead, it is handled by the shared library. In order to preserve the semantics of the root directory ("/"), however, a dummy directory named **Jade** is created under the root directory. The current implementation does not allow symbolic links across the boundary between these two heterogeneous name spaces. This is because the switch between the Jade File System and the original file system is installed in the user-level shared library rather than in the kernel.

Most Unix-like file systems support the notion of a current directory. With this notion, files/directories can be named either by the *full pathname* (a pathname starting from the root) or by the *relative pathname* (a pathname relative to the current directory). However, the notion of current directory is maintained inside the kernel: The kernel keeps the current directory of each process in the process context (i.e., *u* area)[Bach86]; the system call chdir is used to change the current directory; and the current directory is inherited from the parent when a process is forked. Because it is implemented at the user-level, the Jade Library maintains the current directory as an environment variable. All relative pathnames are converted to the full pathname before passing them to the Name Space Manager for

pathname resolution. Environment variables are inherited by the child process from its parent process.

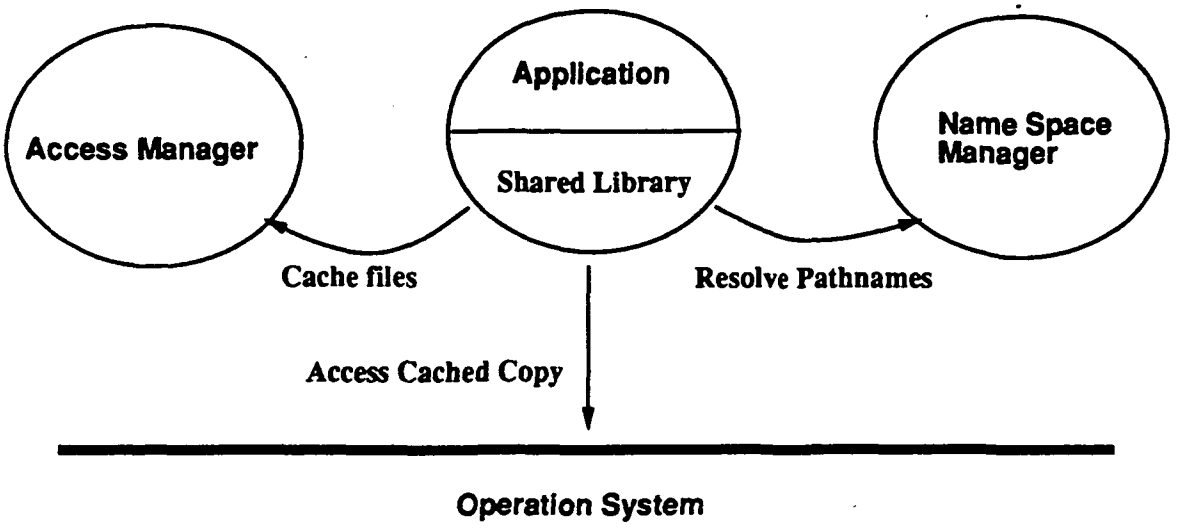


Figure 5.4: Shared Library

Figure 5.4 shows the relationship of the Shared Library with the Name Space Manager, the Access Manager, and the Operation System. Consider now in more detail the library function **open** defined in the Shared Library; the code is given in Figure 5.5. First, the given *path* is translated to a full path *pathname*. The real system call open (through the function **syscall**) is immediately invoked if the *pathname* is in the original file system rather than in the Jade file system; users pay insignificant cost (the cost of a single **if** statement) to access files not in Jade. If *pathname* is in Jade, on the other hand, the service **Lookup** supported by the Name Space Manager is called to resolve the *path*. It returns a reference to the desired file. The **Request** of the Access Manager is then invoked using the reference as an argument. It returns a handle of the cache copy in the cache server. With this handle, the system call **syscall** is invoked to open the cached copy for access.

```
open(path, flags, mode)
    pathname := Relative2Full(path);

    if pathname does not have the prefix "/Jade/" then
        return syscall(SYS_open, pathname, flags, mode);
    fi

    if flags is open_for_read then
        reference := NameSpaceManager.Lookup(pathname);
        if reference is not nil then
            handle := AccessManager.Request(reference, request_for_read);
            if handle is not nil then
                return syscall(SYS_open, handle, flags, mode);
            fi
        fi

    fi
    if flags is open_for_read_only then
        return fail;
    fi

    /* Handling opening a file for write. */
```

Figure 5.5: Modified Function **open**

5.2 Performance

We measured the performance of this prototype with the Andrew Benchmark developed at CMU by M. Satyanarayanan[Howa88]. The input to the benchmark is a read-only source subtree consisting of about 70 files. These files are the source code of an application program and total about 200 kilobytes in size. There are five distinct phases in the benchmark:

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.

Two cases are tested: a local area network and the internet. For each case, we compare the performance of Jade and NFS. 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, while 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 within the universities to 1.544 Mbps T1 connection between these two universities. The client workstation, where Jade and the Andrew Benchmark were running, is a Sun 4/60 workstation with 16 Mbytes main memory and 320 Mbytes disk dedicated for files caching by the Access Manager. The client workstation is running Sun OS 4.1. Both file servers provide Sun NFS protocol for file access.

The performance results of both tests are given in Table 5.4. 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[1]. Since the

[1]The consultation to the Access Manager is omitted since files are located on the same host as the Access Manageris located.

	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 5.4: Performance Results

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. The paper [Welc89] shows 33-41% slowdown when running the Andrew Benchmark.

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's time. The major drawback of this access pattern is that the cost of copying files is extremely high. As discussed in Chapter 4, Jade provides a new function **Relabel** to let users change the reference associated with the cached copy from its source to a new sink. 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 that of 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, 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 5.4 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 that of NFS is 50 seconds. 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. 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 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's time (344 seconds versus 568 seconds). This result is extremely important because the majority of file access in a research or academic environment invokes viewing, editing, and compiling a small set of files[Floy86b].

5.3 Discussion

The main goal of this thesis is to design a file system that is scalable, as well as practical for an internet environment. This section summarizes lessons we have learned in achieving this goal. Although the design decisions were made based on this particular problem domain, most of them can be applied in general to the design of large distributed systems.

Naming Conventions vs. Naming Systems

Distributed systems such as Plan 9[Pike90][Pres91] and Cellular Andrew[Ever90][Zaya88] have naming conventions explicitly built into the system. Thus, the implementation of the system relies heavily on these conventions in that they must be followed by each component in order to compose the whole system. For example, the Cellular Andrew Environment requires that the Andrew file tree be rooted as **/afs** in each autonomous unit (called a *cell*). Each cell owns one node under this root directory, and the cell's individual file systems can only be mounted under this cell node. As another example, Plan 9 defines a set of pathnames with special meaning, e.g., **/proc/77/mem** for the virtual memory of process number 77.

While such techniques simplify the design and implementation of the system, built-in naming conventions restrict the scalability and flexibility of the system. We believe that naming conventions should be independent of the design and implementation of a naming system and should be defined by users. Jade presents users with a fundamental abstraction (logical name spaces) and basic tools (mount operations) and lets users build their own custom naming environments. The result is that Jade provides more freedom than other systems for users to tailor their own naming environment.

Global Name Space vs. Per-User Name Space

The concept of a name space being global was introduced by Multics[Orga72][Salt78] and widely adopted by most Unix-like systems[Ritc78]. The advantage of this concept is that it supports a coherent view among users and hence promotes resource sharing. However, a global name space implies the existence of *central control*, and central control is not amenable to scalable systems. The Amoeba File System[Tane90][Mull85], for example,

uses a central directory server to support the global name space. This central server would be a performance bottleneck in a large scale system. Cheriton and Mann[Mann87][Cher89] introduce *decentralized naming* in which the naming hierarchy is partitioned into global directories, regional directories, and local directories. This design suggests a possible solution for constructing a global name space. 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 and the lack of flexibility to tailor the name space for individual needs.

Jade completely decentralizes the construction and maintenance of name spaces from system administrator to individual users. The scope and complexity of a per-user name space are less than a global name space. This is because although the number of available file systems in an internet is huge and growing dynamically, the number of file systems an individual user wants to access at one time remains small and relatively stable.

In contrary to the *central control* method, Levy and Silberschatz have suggested a *cluster* model that partitions a system into a collection of semi-autonomous clusters[Levy90]. Each cluster is well balanced so that it can be used as a basic modular building block to scale up the system. Jade exemplifies this cluster model. Each per-user file system is a cluster that consists of a set of physical file systems and a dedicated cluster server (i.e., the Name Space Manager and the Access Manager), which can operate independently. A collection of per-user file systems, however, can be joined together by mounting one another to create a bigger and bigger global system. Chapter 6 presents an example of a global environment built on top of the Jade file system.

Local Resolution vs. Remote Resolution

Chapter 3 describes two resolution methods—local resolution and remote resolution—that can be used to interact with mounted file systems in order to resolve a path. Like Locus and Andrew, Jade adopts the local resolution method and caches directory entries. This decision is based on the observation 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.

The Network File System[Sun86a] uses remote pathname resolutions to allow systems to use different way to resolve a name. In particular, it supports access to file systems located on personal computers running the DOS operating system. Jade focuses on a different application domain—an internet—where many available file systems support a Unix-like, tree structural naming space. While the syntax of pathnames in systems such as VAX/VMS may differ slightly from that of traditional Unix file systems, the difference can be hidden inside the access protocols.

Iterative vs. Recursive Pathname Resolution

The two methods to resolve a pathname on sequences of name spaces presented in Chapter 3 are the recursive method and the iterative method. The recursive method has the advantage of completely hiding forward mounts from the current name space. Therefore, the procedure for handling logical file system mounting is treated in exactly the same way as that of the 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.

On the other hand, the iterative method exposes skeleton directories. Because the original name space has full control over the resolution procedure, it is easy to detect loops in the mounting graph.

Another advantage of the iterative method is that it is easy to handle authentication control. Consider a sequence of name spaces N_0, N_1, N_2, and N_p. Let N_0 be the name space receiving the user's query. Assume N_0 refers to N_1 which in turn points to N_2, and so on; N_p is the physical file system where the desired files are located. Because the

original name space N_0 maintains the inquirer's authentication information on a per-host basis, N_0 can issue the proper authentication to N_p. The recursive method needs an extra pair of messages between N_2 and N_0 to get the authentication information before N_2 can query N_P on behalf of the inquirer. When mounting relationships becomes more complicated (e.g., multiple mounts), this overhead becomes even worse.

Remote Server vs. Local Context

In Plan 9[Pike90][Pres91], the name space is implemented as one of the process's *contexts*. The purpose of this design is to provide a virtual machine for each process in order to support heterogeneous environments. However, whenever invoking a new job on other servers, it needs to re-construct a new naming environment for the newly created process. Jade's naming scheme also is able to support the concept of per-process name spaces. However, Jade implements a name space as a *name server* rather than as a context associated with an object (e.g., process). It trades the cost of querying a separate name server for the cost of generating a new naming environment at fork time.

Symbolic Links, Directories, and Skeleton Directories

Most Unix-like file systems support the notion of symbolic links to let users tailor the global name space. Like a skeleton directory, a symbolic link redirects a path from one subtree to another. Unlike a skeleton directory, a symbolic link is applied only within the file system in which it resides, and it 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.

The notion of skeleton directories is a generalization of symbolic links and directories; directories and symbolic links are two special cases of skeleton directories. Figure 5.6 sketchs 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**. In

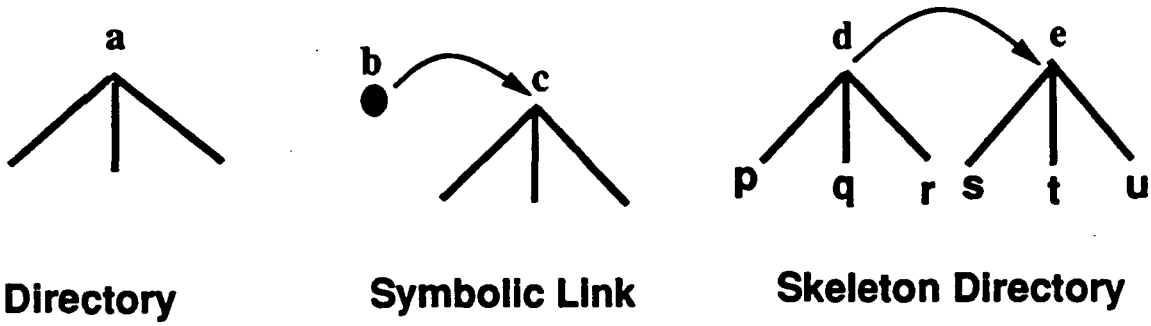


Figure 5.6: Directories, Symbolic Links, and Skeleton Directories

fact, the GetSkeletons function returns local entries under a given pathname in a logical name space. 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**. The node referred to by a skeleton directory can be either a symbolic link, a directory, or another skeleton directory.

Multiple Mounts vs. Union Mounts

Korn and Krell's 3-D file system[Korn90], Sun's Translucent File Service (TFS)[Hend90], and Neuman's Prospero[Neum89] 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.

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 5.7. 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).

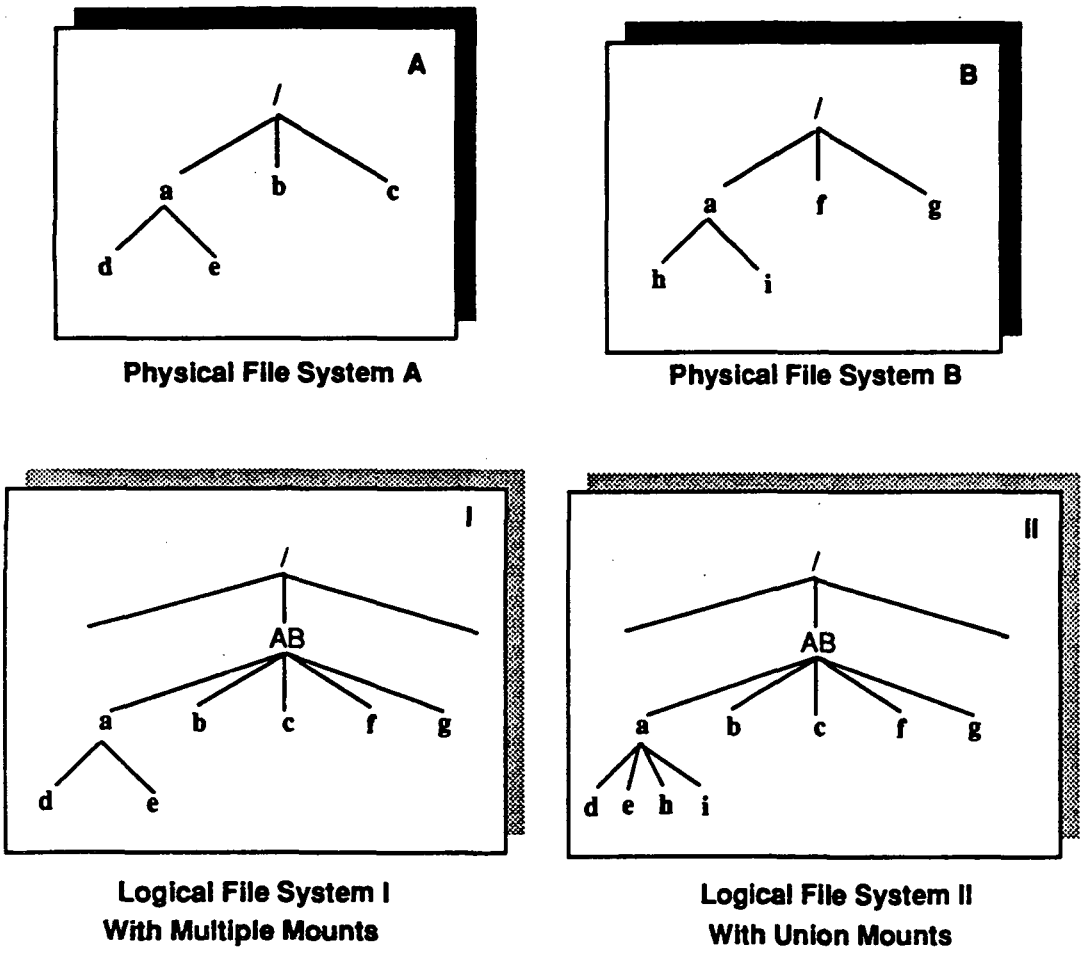


Figure 5.7: Comparison of Multiple Mounts and Union Mounts

Both multiple mounts and union mounts are equally functional. 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, 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 file attributes in the union mount case; it is not clear where the attributes are recorded.

Second, it is more expensive to resolve a pathname in a name space with union mounts. This is because whenever failing to resolve a name in one directory, it needs to *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.

Caching Entire Files

The Cedar file system[Giff88][Schr85] developed by Xerox Palo Alto Research Center 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[Sun86a]. In Jade, caching entire files is essential to access internet files, as described in the previous section. There are two reasons. 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.

There is one potential problem with this approach: access to very large files. Jade provides a ***partial*** solution for this problem. It allows users 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 in order to access larger files. Consistency among cached copies of a file is another problem.

The DEcorum file system[Kaza90] introduces a token mechanism to solve this problem. Chapter 7 suggests future research in this area.

CHAPTER 6

APPLICATIONS

Jade provides a rich set of naming facilities, including per-user logical name spaces, the ability to mount logical name spaces, multiple mounts, and Name Space Stacks. These facilities are not only useful to access internet files, but are also applicable to a variety of other uses. This chapter illustrates these novel features from the application's perspective by presenting the five example uses. The first example shows that Jade provides a rich set of functions that allow users to tailor their private name spaces to fit their personal needs. The second example presents a new method to download software from the internet using the concept of the Name Space Stack and multiple mounts. The third example illustrates how the mount mechanism can be used to build an architecture-specific name space in a heterogeneous environment. The fourth example describes a version control mechanism built on top of Jade. This mechanism provides a hierarchical view of a collection of files for each programmer and allows maximal sharing among these files. The last example illustrates a global, internet-wide name space that is built on top of Jade without any modification to the file system.

6.1 Overview of Jade's Features

Jade provides a rich set of naming facilities. This section re-examines them, with an emphasis on how applications can take advantage of these schemes. In summary, Jade introduces the notion of fine-grain logical name spaces as a new dimension of locating files in the file system, in addition to directories and filenames. It enhances the mount operation so that multiple file systems are able to group together. It also invents the concept of Name Space Stacks to let users manipulate multiple logical name spaces.

Jade extracts the notion of the logical name space from the physical construction of file systems. It allows users to form their own views of a collection of file systems by

constructing their private logical name spaces. In fact, Jade supports name spaces in 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. In addition to directories, the notion of logical name spaces offers a new method for users to organize their files. 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.

Jade allows a logical file system to be mounted into other logical file systems just like a regular physical file system. In addition to promoting file sharing among users, this feature provides a mechanism to overlap multiple name spaces to have a mixed view among them. As in the previous example of the large software project, it is possible to overlap three name spaces: the one for shared codes, the one for the Sparc architecture, and the one for Sun OS 4.1., in order to have a view of the project for this particular software and hardware configuration.

With multiple mounts, Jade allows users to mount multiple file systems on a directory. There are several occasions where these features are very useful. For example, multiple mounts are capable of supporting the same function provided by the notion of *search path* in Unix. Section 6.2 describes this feature in more detail, and Section 6.5 applies it into a version control mechanism. As another example, users can put a local, writable file system in front of a remote, read-only file system on which source files are located. As a consequence, users are able to transparently read the latter file system while writing output on the former file system. Section 6.3 applies this feature to download software from the internet.

Jade supports Name Space Stacks as a simple mechanism to let users manage multiple logical name spaces. In addition, the Name Space Stack is applicable to other uses. For example, it is possible to *checkpoint* and *rollback* on mount operations using the Name Space Stack. That is, the current view of the logical file system can be saved by pushing a

new logical name space on top of the stack. Subsequent mount operations would only affect the newly created name space. The view can be rollbacked to the saved one simply by popping off the topmost logical name space in the stack. As another uses, with the Name Space Stack, users are able to switch back and forth to different logical name spaces in order to perform different tasks. This feature is similar to the directory stack mechanism provided by csh in Unix, where users are able to pop the stack to go back to the previous directory. Furthermore, each name space in the stack is capable of including as well as hiding information, like a translucent paper. The view of a Name Space Stack, therefore, is the result of overlapping a stack of translucent papers. For example, when running a text processing application, it is possible to overlap the application-specific name space and the invoker's name space, and resolve naming in both of them.

6.2 Tailoring a Private Name Space

Because 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 hierarchies, and thus *customize* systems according to their own preferences. There are several ways that users can tailor their name spaces. Figure 6.1 illustrates an example of a private name space used in this section.

First, skeleton directories might not be part of any physical file system and may serve only as logical directories with entries of other skeleton directories. These skeleton directories are created by the mount operation with the null option. For example, the directory **/jade/doc** in Figure 6.1 is not contained in any physical file system. Also, the resolution procedure implies that the name of a skeleton directory has preference over the names of files/directories in physical file systems, such that names in the private hierarchy *supersede* names in the underlying file systems. For example, if there were a directory named **/usr/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/jade/doc**. Additionally, had there been a file or directory named **/usr/jade/junk** on host **meg**, then it would have been hidden by an opaque node named **junk** in the logical name space. Because the opaque node is bound to no file

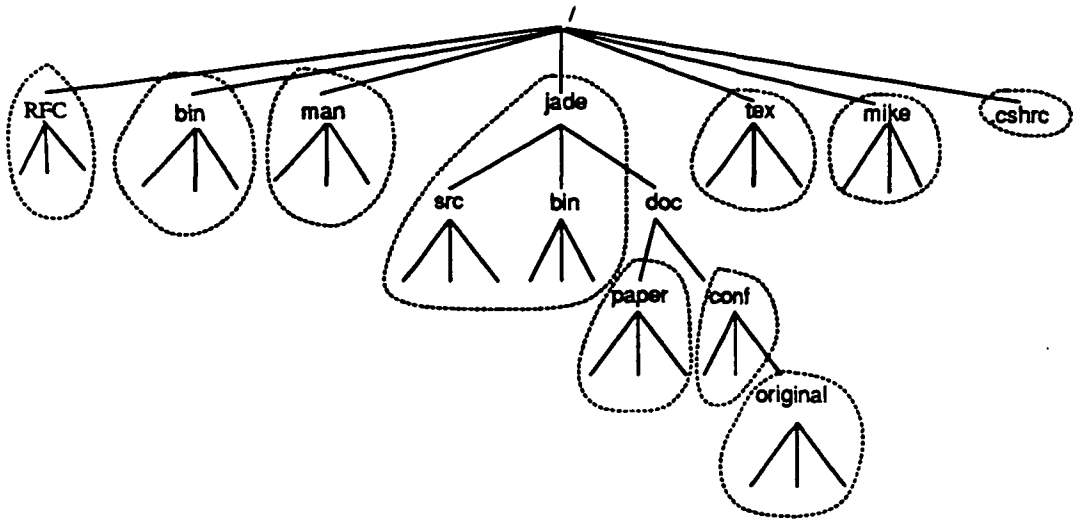


Figure 6.1: Private File Hierarchy

systems, its only purpose is to *hide* something in the underlying file system.

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[1].

Finally, the multiple mount option can be used to locate a file that is replicated in several file systems. If a failure causes one physical file system to become unreachable during pathname resolution, Jade consults the next physical file system in

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

the reference list. For example, the directory **/man** points to **jag:/usr/share/man** and **meg:/usr/share/man**; the file **.chsrc** points to **jag:/usr/john/.cshrc** and **meg:/usr/john/.cshrc**. Note that the physical file systems under the directory are identical and read-only. Putting them under the same name makes the replication property transparent to users.

6.3 Downloading Software from the Internet

Jade supports multiple mounts to allow more than one file system to be mounted onto a directory, each of which may provide a different access protocol. Also, the concept of the Name Space Stack supports the rollback function on mount operations. This section describes a novel method to download software from the internet using these two techniques.

Suppose users want to install the software **grep** from the Free Software Foundation(GNU); the sources are located on the directory **pub/gnu/grep** in the host named **prep.ai.mit.edu**[2].

Installation includes the following five steps. The first step is to create a new name space to handle this task. That is,

```
% NSPush
```

creates a new name space and pushes it on top of the user's Name Space Stack. Subsequent mounting operations affect only this newly created name space. However, because its root points to the name space underneath it, the view of the Name Space Stack remains the same.

The second step is to mount the source file system by the command

```
% mount /grep UFS:jag:/tmp FTP:prep.ai.mit.edu:pub/gnu/grep
```

where the first argument **/grep** is the pathname of the skeleton directory in the logical name space where file systems are mounted, and the second and third arguments specify

[2]Actually, sources in **prep.ai.mit.edu** are stored in a compressed tar file. Chapter 7 suggests mechanisms to mount file systems in the compressed tar format; the system would automatically extract files whenever they are visited. In this example, however, we assume that sources have been extracted from the tar file.

references to two physical file systems. The first reference refers to a directory **/tmp** on the local machine (**jag**), which is accessed by the protocol UFS, while the second reference specifies the source file system (**prep.ai.mit.edu:pub/gnu/grep**), which is accessed by the protocol FTP. Notice that according to the semantics of multiple mounts described in Section 3.2.3, new files are created on the first mounted file system. Because the source file system is read-only and located in the internet with high cost to access, using a local temporary file system as a "work sheet" is essential in this example.

The third step is to generate object files with the regular method as follows:

```
% cd /grep
% make
```

The command **make** uses files in the source file system as input and generates temporary files and object files in the first mounted file system.

In the fourth step, the object file is installed in a proper place, e.g., the directory **/bin**, as follows:

```
% cp /grep/grep /bin
```

Finally, the command

```
% NSPop
```

is used to remove the name space of the top of the Name Space Stack, and as a consequence, the mount operation in the second step is undone.

In summary, this example demonstrates two interesting features. First, with the Name Space Stack, users can easily undo mount operations and rollback to the previous view. Second, with multiple mounts, a local, writable file system is put in front of a remote, read-only file system. Thus, users are able to transparently read the latter file system while writing output in the former file system.

6.4 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 who uses either a MIPS or a Sun SPARC workstation, 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 directories pointing to the proper directories in the name space **main** as illustrated in Figure 6.2. 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 name space **sparc** (**mips**) has two skeleton directories: root "/" pointing to the root of the name space **main**, and **/bin** pointing to **/bin/sparc** (**/bin/mips**) of the name space **main**. When the user logs onto the workstation, the appropriate name space (either **mips** or **sparc**) is initiated automatically,[3] and the user can use the same name to address the binary regardless of which of the two workstations he or she is using.

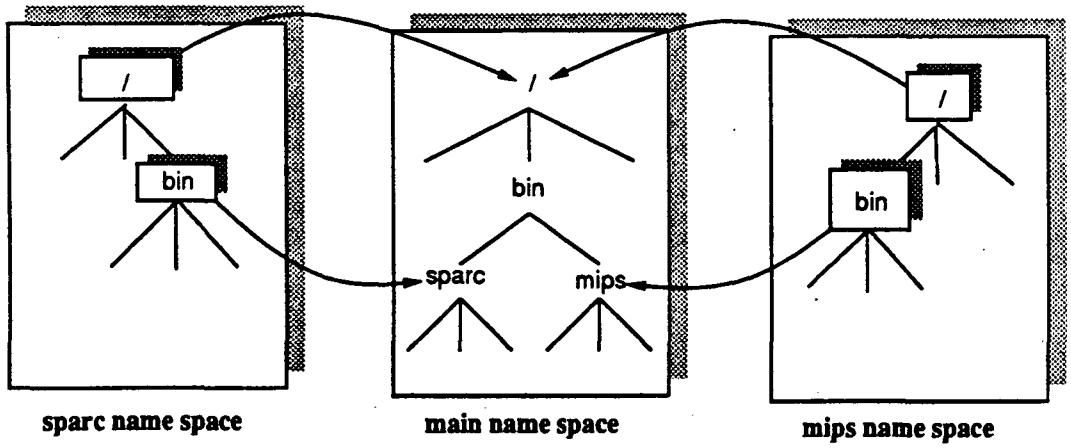


Figure 6.2: Architecture-Dependent Name Spaces

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

This example takes advantage of the fact that a skeleton directory may contain other skeleton directories as its children. That is, although the root of **sparc** refers to the root of **main**, it contains a skeleton directory **/bin**. Therefore, the directory **/bin** in **main** is completely hidden from users, and **/bin** in **sparc** refers to **/bin/sparc** in **main** instead. Finally, it is worth noting that it is impossible to implement this example using symbolic links.

6.5 Version Control

This section discusses how a version control mechanism can easily be built on top of the Jade file system. 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 on different versions and switch between versions easily. Second, the system should provide a mechanism to let each user build a private working area, without worrying about interference from other programmers. Traditional version control 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 is 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 that the multiple references associated with one skeleton directory provide a hierarchical view of a set of files located in different physical directories (even on different hosts). For example, Figure 6.3 shows a software development environment in which two programmers, John and Mike, share files located in different versions. There are four versions located on different file systems: **meg:/jade/release_version/src** for the release version,

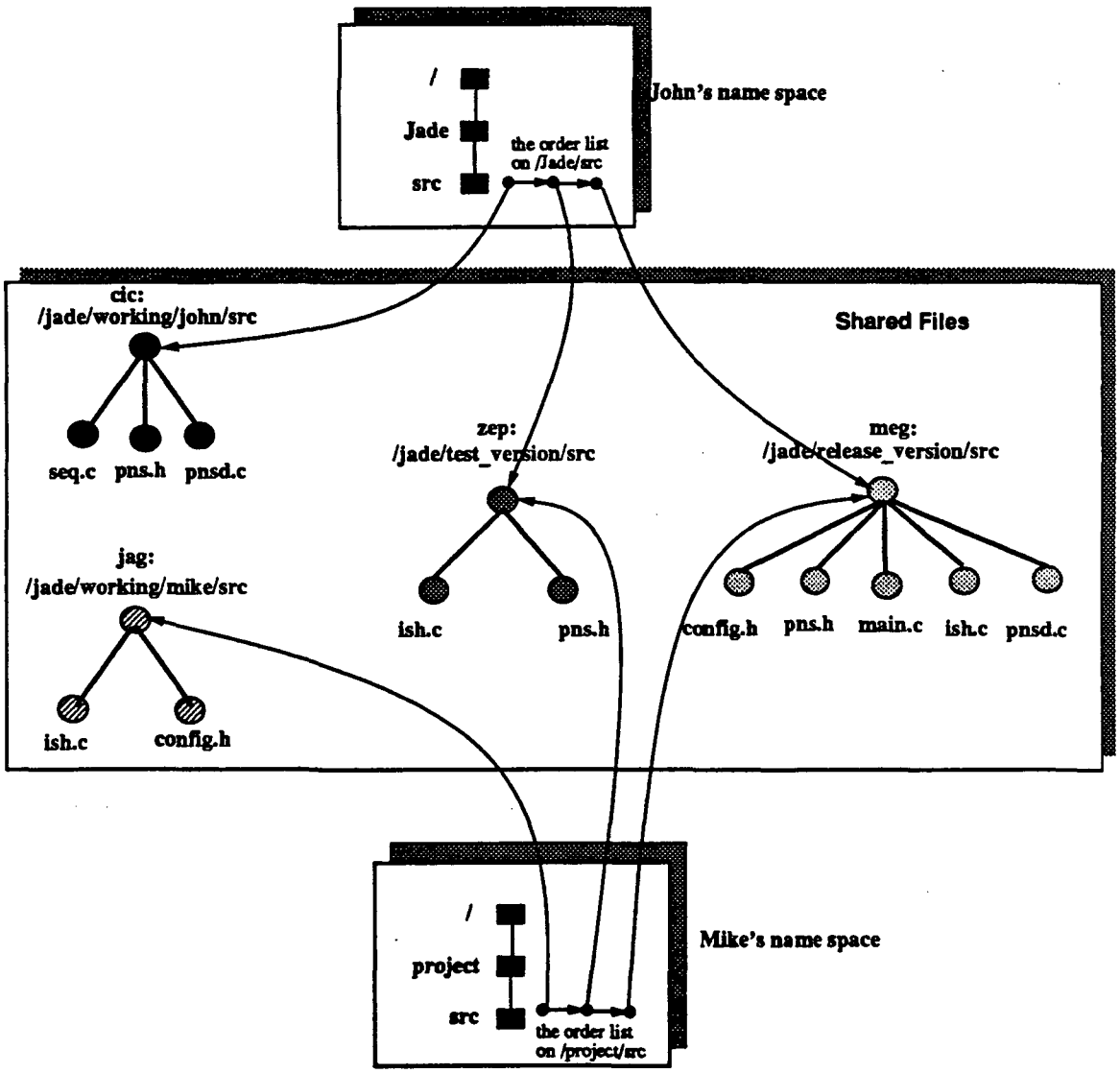


Figure 6.3: Software Development Environment

zep:/jade/test_version/src for the testing version, **cic:/jade/working/john/src** for John's working version, and **jag:/jade/working/mike/src** for Mike's working version. Both John and Mike have their own private name spaces. Consider the skeleton directory **/Jade/src** in John's name space, which consists of three references: The first points to **jag:/jade/working/john/src**, the second refers to **zep:/jade/test_version/src**, and the last points to **meg:/jade/release_version/src**. John's resulting hierarchy is shown in Figure 6.4.

The system also lets 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.

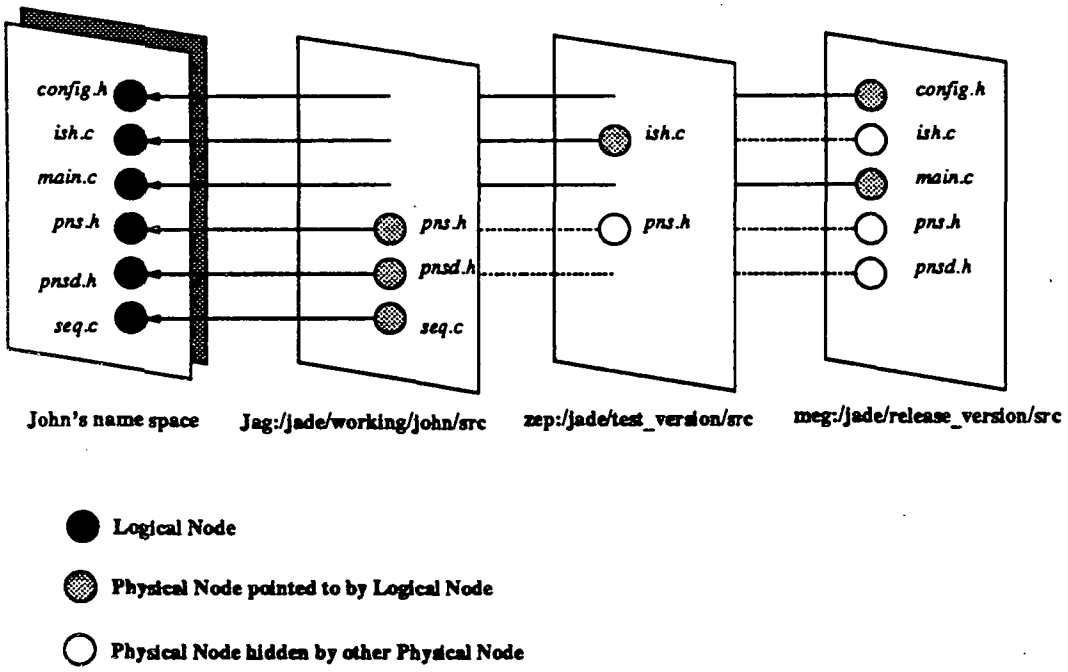


Figure 6.4: Overlaid View

Checking in and out files from different versions located on different file systems is very straightforward in Jade. By mounting the file system where the version is located on a temporary directory, users are able to access one particular version directly. Users can then copy files with the regular command (e.g., **cp**). For example, Mike can check out the

file **main.c** from **meg:/jade/release_version/src** by issuing the following sequence of commands:

```
% mount /tmp_mount meg:/jade/release_version/src
% cp /tmp_mount/main.c /project/src/main.c
% unmount /tmp_mount
```

Indeed, the physical file system where a file is located can easily be found by the command **"ls -l"**, and therefore, the command **checkin** and **checkout** can be implemented by simple shell scripts.

Korn and Krell's 3-D File System[Korn90] and Sun's Translucent File Service (TFS)[Hend90] are designed for software development. Both 3-D and TFS use the *viewpath* mechanism to support union mounts. Section 5.3 have compared union mounts with Jade's multiple mounts. Both 3-D and TFS also provide a *copy-on-write* semantics in that only the first file system defined in the viewpath is writable; all other file systems are read-only. Consequently, when users modify a file located on the file system other than the first one, the file is copied to the first file system before it is modified. The drawback for the copy-on-write semantics is that when intermediate directories of the visited file do not exist in the first file system, the system needs to create each of these directories in the first file system before the writable copy can be made. It would be very expensive to access files with long pathnames. Jade does not support the copy-on-write. This is because Jade is designed for general use, not just for software development.

6.6 Global Name Space in Jade

Jade's naming scheme is able to support the construction of an internet-wide, global name space. This section presents an example of a global name space that is built on top of the Jade file system without any modification to Jade. This global name space glues together a collection of logical name spaces by introducing a set of naming conventions. All logical name spaces in the environment are organized into three layers. The first layer consists of logical name spaces belonging to individual users, called *principal* name spaces. The second layer includes a set of backbone name spaces, called *cell* name spaces, for individual autonomous administration units (e.g., departments). The third layer has only one backbone name space, called the *root* name space, including all cell name spaces. Figure 6.5 presents one instance of this global name space.

Each cell name space is addressed by its domain name[Mock87]. For example, the cell name space with the name **cs.arizona.edu** is the backbone name space for the Department of Computer Science at the University of Arizona, while the cell name space **cs.purdue.edu** is for the Department of Computer Science at Purdue University. Each cell name space mounts all principal name spaces available in the local site onto skeleton directories under the root (/). As in this example, the pathname **/mike** in the cell name space **cs.arizona.edu** refers to the root of the principal name space for Mike, while the pathname **/john** points to John's name space. Each principal name space, on the other hand, mounts its cell name space onto the skeleton directory **/@**. Therefore, the pathname **/@/john/foo** in Mike's name space points to the same file as the pathname **/foo** in John's name space. In fact, pathnames with the prefix as

/@/principal_name

are considered **cell** pathnames and have global meaning within the cell. That is, the pathname **/@/john/foo always refers to the same file regardless of what principal name** space in the cell is used. Also, principal name spaces can make a ***shortcut path*** by mounting the root of their name space under the directory **/@**. For example, the skeleton directory **/@/mike** in Mike's name space refers to the root of his name space, and therefore, pathnames starting with **/@/mike** in Mike's name space can be resolved within Mike's

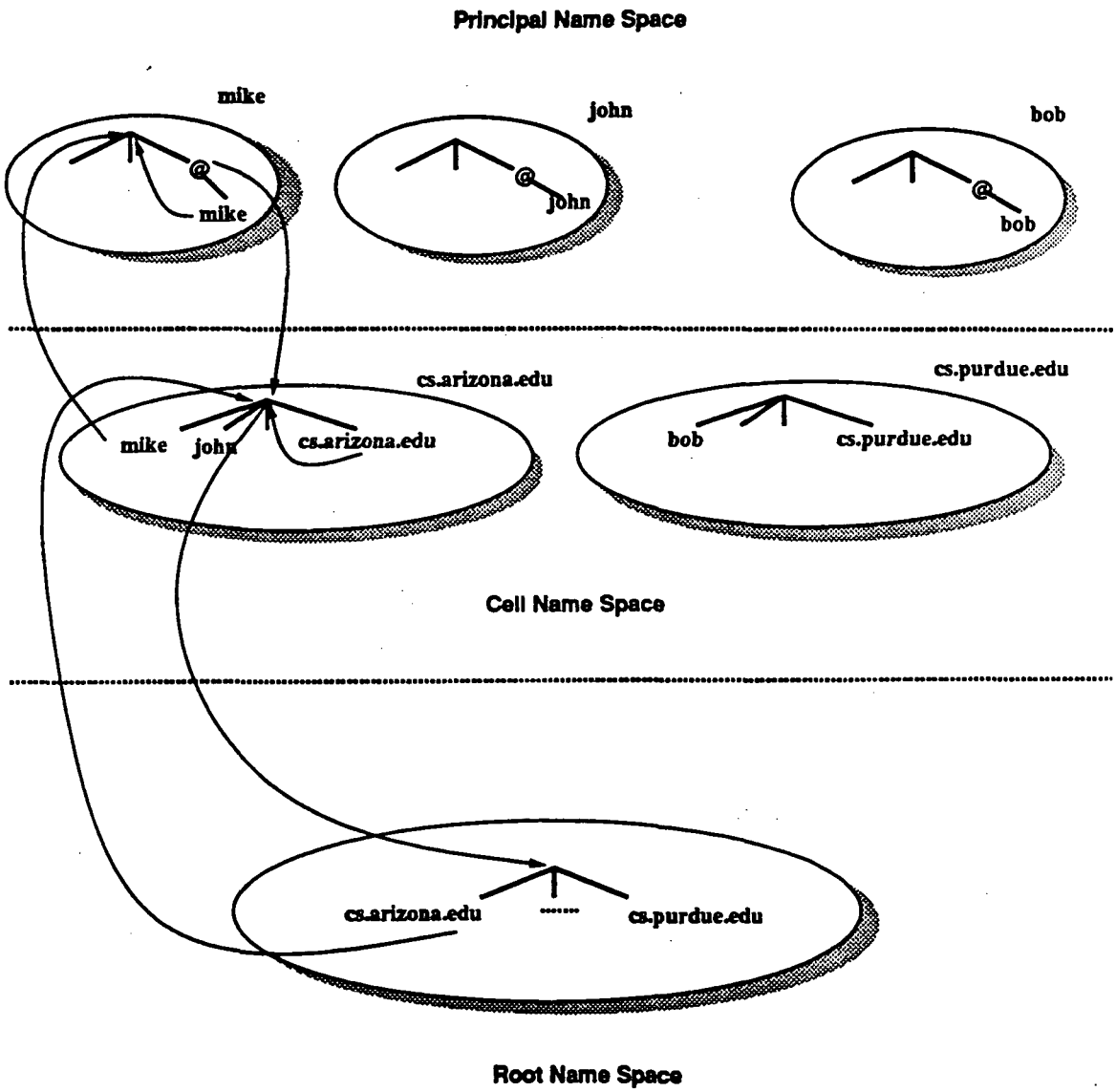


Figure 6.5: Global Name Space in Jade

name space without consulting the cell name space at all.

The root name space includes cell name spaces available in an internet. It mounts cell name spaces under the root directory. Each cell name space mounts the root name space onto its root (/). A *global* pathname is a pathname with the prefix as follows:

/@/cell_name/principal_name

and it refers to a principal name space *principal_name* in the cell named *cell_name*. A global pathname refers to the same file regardless of what principal name space is used. Like the shortcut path used in principal name spaces, cell name spaces mount themselves under the root directory. That is, the skeleton directory **/cs.arizona.edu** in the cell name space **cs.arizona.edu** points to its root.

Consider resolving the following five pathnames from Mike's name space. In particular, we focus on the logical name spaces in the calling sequence in order to resolve a given pathname. Table 6.1 summarizes the discussion. The first pathname **/src/bar**, which is a regular pathname, can be resolved in Mike's name space without invoking other logical name spaces. The second pathname **/@/mike/src/bar**, which is a principal pathname, still can be resolved within Mike's name space because of the shortcut path **/@/mike** in Mike's name space pointing to its root. The third pathname **/@/john/foo** points to a node in John's name space; logical name spaces in the calling sequence include Mike's name space, the cell name space **cs.arizona.edu**, and John's name space. The next pathname **/@/cs.arizona.edu/john/foo** is a global pathname of the previous pathname from Mike's name space, but because of the shortcut path **/cs.arizona.edu/** in the cell name space **cs.arizona.edu**, consulting the root name space is avoided and the same sequence of logical name spaces as in the previous example is invoked. Finally, the pathname **/@/cs.purdue.edu/bob/bar** is a global pathname and it needs to invoke Mike's name space, the cell name space **cs.arizona.edu**, the root name space, the cell name space **cs.purdue.edu**, and Bob's name space.

In summary, this design introduces a *bottom-up* naming scheme in that the path to resolving a pathname starts from the bottom of the global naming tree—the principal name space. It walks to the middle—the cell name space—and to the top—the root name space.

Pathname	Logical Name Spaces invoked
/src/bar	Mike
/@/mike/src/bar	Mike
/@/john/foo	Mike, cs.arizona.edu, John
/@/cs.arizona.edu/john/foo	Mike, cs.arizona.edu, John
/@/cs.purdue.edu/bob/bar	Mike, cs.arizona.edu, root, cs.purdue.edu, Bob

Table 6.1: Pathname Resolutions in the Global Name Space

It may go down either from the cell name space to a principal name space, or from the root name space to a cell name space and to a principal name space. In comparison with this bottom-up method, the naming schemes suggested by Cheriton and Mann's Decentralized Naming[Mann87][Cher89] and the Cellular Andrew Environment[Zaya88] are *top-down* in that pathname resolutions always start from the root of the global naming tree. The major advantage of the bottom-up naming is its high locality in that the majority of pathname resolutions can be done in principal name spaces without invoking cell name spaces or the root name space.

CHAPTER 7

CONCLUSIONS

This dissertation presents the design and implementation of the Jade file system, which provides a uniform mechanism to name and access files in a heterogeneous internet environment. This chapter summarizes the contributions and discusses future research.

7.1 Contributions

Most distributed file systems fail to scale from local area networks to an internet. This thesis identifies four characteristics of scalability: size, wide area, autonomy, and heterogeneity. 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 in order to scale well in practice. Finally, heterogeneity is the nature of an internet file system not only because of its size but also because of its autonomous property.

The primary goal of this research is to design a file system for the internet environment that is both *scalable* and *practical.* In order to achieve this goal, we have designed, implemented, and evaluated the Jade file system. The naming scheme invented for Jade not only is useful to access internet files, but also is applicable to a variety of applications.

7.1.1 Jade is Scalable.

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, this research focuses on accommodating exist-

ing distributed file systems. Particularly, Jade emphasizes the integration of heterogeneous file access protocols.

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. This feature not only supports a simple method to facilitate file sharing, but also provides a tool to link logical file systems.

By mounting other file systems, a collection of logical file systems can be joined together to form a bigger, global system. The relationship among all logical file systems is, however, arbitrary and voluntary without central authorities, specific configurations, or any kind of built-in naming conventions. Chapter 6 presents an example of a global system that is built on top of Jade without any modification to the file system.

7.1.2 Jade is Practical.

Jade is practical in two respects. First, Jade provides complete autonomy. It is designed under the restriction that the software and administration policy of the underlying physical file systems may not be changed. The underlying physical file systems treat an instance of Jade as a regular file system user without any special privileges. More precisely, Jade is implemented, as well as installed, on client workstations 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, experiments with the prototype demonstrate that the design of the Jade file system has an acceptable performance. Statistics show that network latency, which is not an issue in local area networks, becomes an important factor of performance in the internet. In Jade's design, we paid careful attention to avoiding unnecessary network messages between clients and file servers in order to achieve acceptable performance.

To reduce network traffic, Jade adopts techniques of caching entire files and local pathname resolutions. For whole file caching, opening a file causes it to be cached in its entirety, on some nearby disk. Reads and writes are directed to the cached copy without involving the original servers. The valid cached copy can be used for further opens as well.

Jade uses the local resolution method and caches directory entries. Because of a high locality of per-user file access patterns, a directory cache has a high hit ratio, and much of the network traffic for importing directory entries from remote file systems is avoided.

7.1.3 Rich Naming Facilities

Jade provides a rich set of naming facilities, including:

- Fine-grain logical name space;
- Mounting logical file systems;
- Name Space Stack;
- Multiple mounts;
- A generalization of a symbolic link and a directory.

In Jade, we extract the notion of logical name spaces from the physical construction of file systems. Jade allows users to form their own views of a collection of file systems by constructing their private name spaces. The ability to mount logical name spaces allows users to overlap a set of logical name spaces in order to have a mixed view among them. It then encourages users to generate multiple name spaces, each of which is dedicated to one special task. Jade provides the Name Space Stack as a simple way to group a set of logical name spaces. With the Name Space Stack, users are also able to perform checkpoint and rollback functions on mount operations. With multiple mounts, multiple file systems can be grouped under one directory. Finally, the notion of the skeleton directory is a generalization of symbolic links and directories. Like a symbolic link, it refers a node to another node; like a directory, it has local entries. Chapter 6 illustrates several examples that take advantage of these naming faclities.

7.2 Future Directions

The preliminary experience on the prototype has shown that Jade is a good start toward an internet-wide file system. We have demonstrated that the design of Jade is scalable as well as practical. However, a full test of Jade would require implementation at several sites with active user communities. The feedback from the users would help in refining the design. In particular, it is very important to understand the patterns by which users access

files located on the internet. File access patterns in local area networks have been studied by Floyd[Floy86a][Floy86b], Ousterhout *et al.*[Oust85], and Satyanarayanan[Saty81]. Characteristics of internet applications in general have been investigated by Caceres *et al.*[Cace91] and Paxson[Paxs91]. We anticipate that the majority of file access in an internet invokes viewing, editing, and compiling a small set of files, whereas commands used to invoke these files are located in local area networks. The study of access patterns would benefit the design of cache mechanisms and therefore improve performance.

Most software located in the internet for public access are stored as compressed archives. In particular, they are in the form of compressed tar files. In order to access this software, users need to make a local copy, uncompress it, and extract desired files out of it. It would be much easier to access such data if the archives could be mounted directly as if they were regular physical file systems. In order to support this function, a new kind of agents, called *format* agents, would be needed in addition to *protocol* agents for access protocols. The new format agent would transform files between different storage formats. Other examples of format agents are agents for SCCS[Allm86] and RCS[Tich85]. The SCCS agent, for example, would handle the process of checking in and out files from a SCCS directory. In order to access a given file system, it could need a protocol agent to interact with the file server and a format agent to extract (add) files from (to) the file server. For example, a file system is accessed by NFS and is stored under a SCCS directory; another file system is accessed by FTP and is stored in a compressed tar file. The challenge of designing format agents is how to incorporate them with protocol agents.

As indicated in Chapter 4, Jade distinguishes between the protocol used to access remote resources and the protocol used to access files in the cache server. This approach could be used to name and access resources other than textual files, such as mailboxes and printers. The cached copy in the cache server is considered as a local *image* for the remote resource. The protocol agent not only handles the transmission between the local image and the source, but also deals with the conversion of formats between them. For example, when *opening* a mailbox, a file-like image is created on the cache server. When *closing* it, however, the agent transforms the local image into a message by appending the proper header, and invokes the mail protocol (i.e., SMTP[Post82]) to deliver the message

to the remote mailbox.

In fact, the abstraction of agents is general enough to support a variety of resources. For example, it would be straightforward to implement an *IPC* agent for interprocess communications. An IPC channel is associated with a named file and put on the name space. When opening a file, the Access Manager establishes the connection and returns the handle of the channel for the **Request** operation. Subsequent operations on files are operations on channels. The Access Manager terminates the channel when the file is closed. This feature is similar to the one suggested by Presotto and Ritchie[Pres90].

Jade caches entire files on nearby disks. Consistency of multiple cached copies located on different cache servers is a problem. In order to support more complicated applications, there is a need for an access control mechanism. The other side of this problem is availability. A file is said to be available if it can be accessed whenever needed, despite machine crashes and communication faults. This property is particularly important to the internet file system because the reliability of an internet is much less than that of local area networks. In order to increase availability, replicating files on different file servers is essential. In general, a single file may have multiple cached copies as well as multiple replicated copies. The ideal access control mechanism should also handle consistency among replicated copies on different file systems with different access protocols.

Finally, the ultimate goal of this research is to study access to internet resources[Hutc89b][Pete90]. Workstation users connected to the internet have access to significantly more resources than are available on local area networks. The NREN, for example, connects users throughout the country to file systems, databases, directory services, information archives, supercomputers, and other special hardware. The availability of resources on such national networks will grow as network bandwidth and connectivity increase. Jade is a good start toward this goal, and it also serves as a vehicle for further study in this area.

APPENDIX A

JADE NAMING PROTOCOL SPECIFICATION

The Jade Naming Protocol (JNP) provides transparently remote access to a Name Space Manager. The protocol is designed on top of Sun's Remote Procedure Call (RPC)[Sun86b] and External Data Representation (XDR)[Sun90]. It is specified using Sun's RPC data description language[Sun90].

```
/* The maximun number of bytes in a name argument. */
const MAXNAMELEN = 255;

/* The maximun number of bytes in a pathname argument. */
const MAXPATHLEN = 1024;

/* The maximun number of bytes in an arguement. */
const MAXLINE = 255;

/* The size in bytes of the opaque file handle. */
const FHSIZE = 32;

typedef string filename_t<MAXNAMELEN>;
typedef string user_t<MAXNAMELEN>;
typedef string path_t<MAXPATHLEN>;
typedef string host_t<MAXPATHLEN>;
typedef string args_t<MAXLINE>;
typedef opaque jdhandle_t[FHSIZE];

/*
* jdstat is returned with every procedure's result.
* JD_OK indicates that the call completed successfully and the result is valid.
* The other value indicates some kind of error occured on the server side
* during servicing of the procedure.
*/
enum jdstat {
    JD_OK = 0,
    JD_ERROR = 1,
    JD_NOUSER = 2,
    JD_NONODE = 3,
```

```
    JD_NOPATH = 4,
    JD_IS_SK = 5,
    JD_IS_N_SK = 6,
    JD_STACK_EMPTY = 7
};

/* The enumeration fsprot defines the access protocol */
enum fsprot {
    JD_UNKNOWN = 0,   /* the type is unknown; */
    JD_JFS = 1,       /* Jade Naming Protocol; */
    JD_UFS = 2,       /* Unix File System; */
    JD_NFS = 3,       /* Network File System; */
    JD_FTP = 4,       /* File Transfer Protocol; */
    JD_AFS = 5        /* Andrew File System. */
};

/*
* timeval is number of seconds and microseconds since midnight 1/1/1970,
* Greenwich Mean Time.
* It is used to pass time and date information.
*/
struct timeval {
    unsigned int sec;
    unsigned int usec;
};

/* jdattr contains the attributes of a file. */
struct jdattr {
    host_t          a_host;
    path_t          a_path;
    jdhandle_t      a_fh;
    fsprot          a_prot;
    u_long          a_mode;
    u_long          a_uid;
    u_long          a_gid;
    u_long          a_size;
    struct timeval  a_atime;
    struct timeval  a_mtime;
};

/* jdpath specifies a path in a logical name space named as user. */
struct jdpath {
    user_t user;
    path_t path;
```

```
};

/* jd2path is used in JD_Rename operation. */
struct jd2path {
    user_t user;
    path_t path1;
    path_t path2;
};

/* jddirargs is used in JD_SetAttr operation. */
struct jddirargs {
    struct jdpath jp;
    struct jdattr at;
};

/* jdmkdir is used in JD_MakeDir operation. */
struct jdmkdir {
    struct jdpath jp;
    int mode;
};

/* jdref specifies a reference to a named file. */
struct jdref {
    host_t          host;
    path_t          path;
    jdhandle_t      fh;
    fsprot          prot;
    struct timeval  timestamp;
};

/* The results of JD_Lookup operation are returned in jdrefres. */
union jdrefres switch (jdstat stat) {
    case JD_OK:
        struct jdref ref;
    default:
        void;
};

/* jdmountargs is used in JD_Mount operation. */
struct jdmountargs {
    struct jdpath jp;
    args_t line;
    args_t mode;
};
```

```
/*
* jdentryres is used for directory entries returned for
* JD_GetEntries operation.
*/
typedef struct namenode *namelist;
struct namenode {
     filename_t name;
     namelist next;
};
union jdentryres switch (jdstat stat) {
     case JD_OK :
          namelist list;
     default :
          void;
};

/* jdskres is used for the results of JD_GetSkeleton operation. */
struct jdsk {
     namelist sklist;
     struct jdref refs<>;
};
union jdskres switch (jdstat stat) {
     case JD_OK :
          struct jdsk sk;
     default :
          void;
};

/* jdattrres is used for the result of JD_GetAttr operations. */
union jdattrres switch (jdstat stat) {
     case JD_OK :
          struct jdattr at;
     default :
          void;
};
```

```
/* Service routines. */
program JD_PROC {
    version JD_VERSION {
        void JD_Null(void) = 0;
        jdrefres JD_Lookup(jdpath) = 1;
        jdattrres JD_GetAttr(jdpath) = 2;
        jdstat JD_SetAttr(jddirargs) = 3;
        jdstat JD_Remove(jdpath) = 4;
        jdentryres JD_GetEntries(jdpath) = 5;
        jdstat JD_MakeDir(jdmkdir) = 6;
        jdstat JD_RemoveDir(jdpath) = 7;
        jdstat JD_Rename(jd2path) = 8;
        jdstat JD_Mount(jdmountargs) = 9;
        jdstat JD_Hide(jdmountargs) = 10;
        jdstat JD_Unmount(jdpath) = 11;
        jdskres JD_GetSkeletons(jdpath) = 12;
        jdskres JD_FSInfo(jdpath) = 13;
        jdstat JD_NSPush(void) = 14;
        jdstat JD_NSPop(void) = 15;
        jdstat JD_NSDump(jdpath) = 16;
        jdstat JD_NSLoad(jdpath) = 17;
    } = 1;
} = 20000201;
```

APPENDIX B
JADE ACCESS PROTOCOL SPECIFICATION

The Jade Access Protocol (JNP) provides transparently remote access to a Access Manager. The protocol is designed on top of Sun's Remote Procedure Call (RPC)[Sun86b] and External Data Representation (XDR)[Sun90]. It is specified using Sun's RPC data description language[Sun90].

```
/* The maximun number of bytes in an arguement. */
const AC_MAXLEN = 255;

/* The size in bytes of the opaque file handle. */
const AC_FHSIZE = 32;

typedef string ac_user_t<AC_MAXLEN>;
typedef string ac_path_t<AC_MAXLEN>;
typedef string ac_host_t<AC_MAXLEN>;
typedef opaque ac_handle_t[AC_FHSIZE];

/*
* ac_stat is returned with every procedure's result.
* AC_OK indicates that the call completed successfully and the result is valid.
* The other value indicates some kind of error occured on the server side
* during servicing of the procedure.
*/
enum ac_stat {
    AC_OK = 0,
    AC_ERROR = 1,
    AC_UNMODE = 2,
    AC_FETCH_FAIL = 3,
    AC_RESTORE_FAIL = 4,
    AC_UNJNODE = 5
};

/* The enumeration fsprot defines the access protocol */
enum fsprot {
    JD_UNKNOWN = 0,   /* the type is unknown; */
    JD_JFS = 1,       /* Jade Naming Protocol; */
    JD_UFS = 2,       /* Unix File System; */
    JD_NFS = 3,       /* Network File System; */
    JD_FTP = 4,       /* File Transfer Protocol; */
```

```
    JD_AFS = 5              /* Andrew File System. */
};

/* The enumeration ac_rq_flag defines the type of AC_Request operation. */
enum ac_rq_flag {
    AC_Request_RD = 0,
    AC_Request_WR = 1,
    AC_Request_RD_WR = 2
}

/* The enumeration ac_rl_flag defines the type of AC_Release operation. */
enum ac_rl_flag {
    AC_Release_none= 0,
    AC_Release_Syn = 1,
    AC_Release_ASyn = 2
}

/*
* timeval is number of seconds and microseconds since midnight 1/1/1970,
* Greenwich Mean Time.
* It is used to pass time and date information.
*/
struct timeval {
    unsigned int sec;
    unsigned int usec;
};

/* ac_ref specifies a reference to a named file. */
struct ac_ref {
    ac_host_t           a_host;
    ac_path_t           a_path;
    ac_handle_t         a_fh;
    fsprot              a_prot;
    struct timeval      a_timestamp;
};


/* ac_refres is used in AC_Request operation. */
struct ac_refres {
    ac_ref attr;
    ac_rq_flag flag;
};
```

```
/* ac_jid refers to a cached copy in the cache server. */
struct ac_jid {
     ac_path_t path;
     int jnode;
};

/* ac_res is used for the result of JD_Request operation. */
union ac_res switch(ac_stat stat) {
     case AC_OK :
          ac_jid id;
     default:
          void;
};

/* ac_label is used in JD_Relabel operation. */
struct ac_label {
     int jnode;
     ac_refres label;
};

/* as_rel is used in JD_Release operation. */
struct ac_rel {
     int jnode;
     ac_rl_flag flag;
};


/*Service routines */
program AC_PROC {
     version AC_VERSION {
          void AC_Null(void) = 0;
          ac_res AC_Request(ac_refres) = 1;
          ac_stat AC_Relabel(ac_label) = 2;
          ac_stat AC_Release(ac_rel) = 3;
          ac_stat AC_Output(ac_path_t) = 4;
     } = 1;
} = 20000202;
```

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.

[Bach86] Bach, M. J. *The Design of the Unix Operating System.* Prentice-Hall, 1986.

[Bara87] Barak, A. and Kornatzky, Y. Design principles of operating systems for large scale multicomputers. In Nehmer, J., editor, *Experiences with Distributed Systems*, pages 104–123. Springer-Verlag, Berlin, 1987.

[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.

[Cace91] Cacerest, R., Danzig, P., Jamin, S., and Mitzel, D. Characteristrics of wide-area TCP/IP conversations. In *Proceedings of the SIGCOMM '91 Symposium*, 1991. To appear.

[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 Droms, R. E. Tilde trees in the Unix environment. In *Preceedings of Winter Usenix*, pages 23–29, January 1985.

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

[Come91] Comer, D. E. *Internetworking with TCP/IP Volume I Principles, Protocols, and Architectures.* Prentice Hall, second edition, 1991.

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

[Floy86a] Floyd, R. Directory reference patterns in a Unix environment. Technical Report TR 179, Computer Science Department, The University of Rochester, August 1986.

[Floy86b] 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.

[Gros86] Grosling, J. SunDew—a distributed and extensible window system. In *Methodology of Window Management.* Springer-Verlag, 1986.

[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.

[Hutc89a] Hutchinson, N. C., Peterson, L. L., Abbott, M. B., and O'Malley, S. RPC in the *x*-Kernel: Evaluating new design techniques. In *Proceedings of the Twelfth ACM Symposium on Operating System Principles*, pages 91–101, December 1989.

[Hutc89b] Hutchinson, N. C., Peterson, L. L., and Rao, H. The *x*-Kernel: An open operating system design. In *Second Workshop on Workstation Operating Systems*, pages 55–59, September 1989.

[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.

[Lede89] Lederberg, J. and Uncapher, K. Towards a national collaboratory. *Report of an Invitational Workshop At The Rockefeller University*, March 1989.

[Leff89] Leffler, S. J., McKusick, M. K., Karels, M. J., and Quarterman, J. S. *The Design and Implementation of the 4.3BSD Unix Operating System.* Addison-Wesley Publishing Company, 1989.

[Lein87] Leiner, B. M. Network requirements for scientific research. Request For Comments 1017, USC Information Sciences Institute, Marina del Ray, Calif., August 1987.

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

[Mann87] Mann, T. *Decentralized naming in distributed computer systems.* PhD thesis, Stanford University, Palo Alto, CA, May 1987.

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

[Mull85] Mullender, S. J. and Tanenbaum, A. S. A distributed file service based on optimistic concurrency control. In *Proceedings of the Tenth ACM Symposium on Operating System Principles*, pages 51–62, Orcas Island, WA, December 1985.

[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.

[Neum89] Neuman, B. C. The need for closure in large distributed systems. *Operating Systems Review*, 23(4):28–30, October 1989.

[NSF89] NSF Network Service Center, BBN Systems and Technologies Corporation. *The Internet Resources Guide*, 1989.

[Orga72] Organick, E. I. *The Multics System: An Examination of Its Structure.* MIT Press, Cambridge, Massachusetts, 1972.

[Oust85] Ousterhout, J. K., Costa, H. D., Harrison, D., Kunze, J. A., Kupfer, M., and Thompson, J. G. A Trace-Driven analysis of the Unix 4.2 BSD file system. In *Proceedings of the Tenth ACM Symposium on Operating System Principles*, pages 15–24, December 1985.

[Paxs91] Paxson, V. Measurements and models of wide area TCP conversations. Technical Report LBL-30840, LBL Computer Systems Engineering Group, 1991.

[Pete90] Peterson, L. L., Hutchinson, N. C., O'Malley, S. W., and Rao, H. C. The *x*-Kernel: A platform for accessing internet resources. *IEEE Computer*, 23(5):23–33, May 1990.

[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, Cambridge, Massachusetts, 1985.

[Post81] Postel, J. Transmission control protocol. Request For Comments 793, USC Information Sciences Institute, Marina del Ray, Calif., September 1981.

[Post82] Postel, J. Simple Mail Transfer Protocol. Request For Comments 821, USC Information Sciences Institute, Marina del Ray, Calif., August 1982.

[Post83] Postel, J. and Reynolds, J. TELNET Protocol Specification. Request For Comments 854, USC Information Sciences Institute, Marina del Ray, Calif., May 1983.

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

[Pres90] Presotto, D. L. and Ritchie, D. Interprocess communication in the ninth edition Unix system. *Software—Practice and Experience*, 20(S1):S1/3–S1/17, June 1990.

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

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

[Salt78] Saltzer, J. H. Naming and binding of objects. In Bayer, R., Graham, R., and Seegmuller, G., editors, *Operating Systems: An Advnaced Course*, pages 99–208. Springer-Verlag, New York, 1978.

[Salt84] Saltzer, J. H., Reed, D. P., and Clark, D. D. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 4(2):277–288, November 1984.

[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.

[Saty81] Satyanarayanan, M. A study of file sizes and functional lifetimes. In *Proceedings of the Eighth ACM Symposium on Operating System Principles*, pages 96–108, December 1981.

[Saty85] Satyanarayanan, M., Howard, J. H., Nichols, D. A., Sidebotham, R. N., Spector, A. Z., and West, M. J. The ITC distributed file system: Principles and design. In *Proceedings of the Tenth ACM Symposium on Operating System Principles*, pages 35–50, December 1985.

[Saty89a] Satyanarayanan, M. Distributed file systems. In Mullender, S., editor, *Distributed Systems*, pages 149–188. ACM Press, 1989.

[Saty89b] Satyanarayanan, M. A survey of distributed file systems. Technical Report CMU-CS-89-116, Carnegie-Mellon University, February 1989.

[Saty90a] Satyanarayanan, M. Scalable, secure, and highly available distributed file access. *IEEE Computer*, 23(5):9–21, May 1990.

[Saty90b] Satyanarayanan, M., Kistler, J. J., Kumar, P., Okasaki, M. E., Siegel, E. H., and Steere, D. C. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers, Special Issue on Fault-Tolerant Computing*, 39(4):447–459, April 1990.

[Scha90] Schatz, B. R. *Interactive Retrieval in Information Spaces Distributed across a Wide-Area Network.* PhD thesis, Department of Computer Science, University of Arizona, December 1990.

[Sche86] Scheifler, R. W. and Gettys, J. The X window system. *ACM Transactions on Graphics*, 5:79–109, April 1986.

[Schr85] Schroeder, M. D., Gifford, D. K., and Needham, R. M. A caching file system for a programer's workstation. In *Proceedings of the Tenth ACM Symposium on Operating System Principles*, pages 25–34, 1985.

[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.

[Side89] Sidebotham, B. Rx: A high performance remote procedure call transport protocol. Technical report, Information Technology Center, Carnegie Mellon University, February 1989.

[Sun86a] Sun Microsystems, Inc., Mountain view, Calif. *Network File System*, February 1986.

[Sun86b] Sun Microsystems, Inc., Mountain view, Calif. *Remote Procedure Call Programming Guide*, February 1986.

[Sun88] Sun Microsystems, Inc., Mountain view, Calif. *Shared Libraries*, May 1988.

[Sun90] Sun Microsystems, Inc., Mountain View, Calif. *External Data Representation Standard: Protocol Specification*, March 1990.

[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.

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

[Tich85] Tichy, W. F. RCS—a system for version control. *Software—Practice and Experience*, 15(7):637–654, 1985.

[Walk83] Walker, B., Popek, G., English, R., Kline, C., and Thiel, G. The LOCUS distributed operating system. In *Proceedings of the Ninth ACM Symposium on Operating System Principles*, pages 49–70, October 1983.

[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.