The Virtual System Model:
A Scalable Approach to Organizing Large Systems

by

Barry Clifford Neuman

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

1992

Approved by____
(Chairperson of Supervisory Committee)

Program Authorized
to Offer Degree____________________________________

Date____

University of Washington

Abstract

The Virtual System Model: A Scalable Approach to Organizing Large Systems

by Barry Clifford Neuman

Chairperson of the Supervisory Committee: Professor Edward D. Lazowska
Department of Computer Science
and Engineering

Naming is critical in distributed systems. Names identify files, services, processors, and users. A name service translates the names of objects to the information needed to access those objects. The growth of distributed systems brings with it an increase in the number of objects to be named. Existing naming techniques are derived from techniques used on centralized systems and are not sufficient for organizing the large number of objects that are becoming available.

This dissertation presents the Virtual System Model, a new model for naming that helps users organize the information available in large systems. The Virtual System Model has four principal features: support for customizable name spaces, tools to help users construct name spaces, support for synonyms, and a method for selecting the appropriate name space when resolving names.

In the Virtual System Model users create customized views of the system, called virtual systems, using tools that allow new views to be specified as functions of other, possibly changing, views. In a customized view, objects that users frequently use have short names while those that are never used are hidden from view. A problem that arises from customized naming is the lack of name transparency: the same name may refer to different objects when used in different name spaces. This problem is addressed by supporting closure: each object has an associated name space, and that name space is used to resolve names that are specified by the object.

The dissertation begins by discussing the naming mechanisms used in existing systems and by highlighting the problems that arise as systems grow. The Virtual System Model is described, and its differences from existing models of naming are discussed. The application of the Virtual System Model to distributed systems is described, including its use organizing and searching for information, selecting processors and services, and specifying security requirements.

A prototype file system named Prospero demonstrates the usefulness of the model. The prototype is used on more than 7,500 systems in 29 countries to organize and explore information available from Internet archive sites worldwide. The design, implementation, and use of the prototype are described.

Table of Contents

List of Figures

List of Tables

Acknowledgements

Over the past ten years I have been fortunate to have worked with some of the most talented individuals in distributed computing systems. I owe much of my initial understanding of and outlook on the field to three people, Jerry Saltzer, Dave Clark, and David Gifford, who together taught 6.033, the undergraduate computer systems class at MIT. Dave Clark served as advisor for my Bachelor's thesis and it is through my work for him that I first developed the skills needed for successful research in distributed systems. Jerry Saltzer subsequently provided me with the opportunity to apply those skills in a real system, Project Athena. It was at Project Athena that I learned the difference between demonstrating an idea in isolation and solving a problem in the real world.

Just as important and just as talented are the individuals I worked with at the University of Washington. I owe a great deal to my advisor, Ed Lazowska, who provided me with additional examples of solutions to problems in distributed systems, and helped me to develop a balanced view of previous work. I also owe him a great deal for standing behind me as I chose my own research topic and for forcing me to narrow the scope of my thesis to that which could be accomplished in a reasonable time. Finally, his comments on numerous drafts of the dissertation greatly improved its content and readability.

Hank Levy, David Notkin, and John Zahorjan were involved from the early stages of my work, asking difficult questions and forcing me to think long and hard about many of the issues. Their input helped me to better understand the problem, and to better understand my own solutions. John Zahorjan's detailed comments on numerous drafts of the dissertation were extremely valuable.

Other members of my advisory committee, including Richard Ladner and Martin Tompa, provided useful feedback on my work through early discussions on the scope of my thesis, later discussions on aspects of the model itself, and by commenting on sections of the dissertation.

Bill Griswold provided help with the implementation of filters, making available a dynamic linker used for one of his research projects, and extending it so that it was suitable for use in Prospero.

Others who provided feedback on the ideas or comments on drafts of the dissertation include Brian Bershad, Andrew Black, David Cohn, Peter Deutsch, Alan Emtage, Barbara Gordon, Bruce Gordon, Terry Gray, Robert Henry, Chuck Kacmar, David Keppel, Alex Klaiber, Mary Ann G. Neuman, Larry Snyder, Karen Sollins, and Alfred Spector.

I'd like to thank the users of Prospero for demonstrating that the problems addressed in this dissertation are real, and that the solutions are useful. I'd especially like to thank Steve Cliffe, Alan Emtage, George Ferguson, Brendan Kehoe, and Khun Yee Fung, who integrated Prospero into their own applications, greatly extending the community of Prospero users. I'd also like to thank Bill Bryant for suggesting the name "Prospero".

Finally, I'd like to thank my father, Peter Neuman, who kept after me these past few years to finish things up, and my mother, Barbara Gordon, who was just as supportive but knew that I didn't need another person constantly asking when I'd be done.

Chapter 1

Introduction

This thesis describes a new model for naming that allows users to cope with the growing size of distributed systems.

Naming is one of the key facilities that unite networked computers into a distributed system. Names are used to identify files, services, processors, and users. The name service translates the names of objects to the information needed to access those objects.

The size of distributed systems is growing. With this growth comes an increase in the number of objects that need to be named. A system's scale has three dimensions and the tasks of naming and finding objects of interest in large systems are complicated by each.

The first dimension is *numerical*. As the number of computers in a distributed system grows, or as the storage available on existing computers increases, the number of objects that are accessible from within the system increases. The number of objects that are of interest to a particular user, however, will not grow as fast as the total number of objects. Thus, for any given user, the bulk of the objects being added are not of interest, and unfortunately, the few that are do not stand out among the clutter of other objects in the system.

The second dimension of scale is *geographic*. In order to achieve acceptable performance, objects are usually stored on or near the systems that frequently access them.

Related information used from different locations is stored separately, and the task of finding information on a particular topic requires looking in multiple places. Unfortunately, it may not be obvious to users where to look.

The third dimension of scale is *administrative*. A large system often spans multiple organizations, each wanting control over "its" part of the system. This control is often granted by giving organizations the exclusive right to assign a subset of the names that are available for naming objects. The result is that a user looking for an object needs to know which organization controls the object, but this information is often unknown.

Existing techniques for naming in distributed systems have evolved from techniques used on centralized systems. These techniques fail to address the problems of organization that arise as systems grow.

1.1 Thesis Statement, Goals and Contributions

My thesis is that existing techniques are not sufficient for organizing information in large systems. As distributed systems grow, so does the number of objects that need to be organized. Existing techniques for organizing information in distributed systems have evolved from the techniques used on centralized systems. These techniques are not sufficient for organizing the large amount of information that is becoming available. Better tools are needed.

Improving the organization of objects of known interest benefits users who are trying to identify objects of potential interest. When information is poorly organized it is difficult for users to identify objects of potential interest. Thus, improving the tools available for organizing information will also make it easier for other users to find information that has been organized with those tools.

Customization plays an important role in the organization of objects. It aids users in keeping track of the objects that they use on a regular basis. It is also possible to make use of the information specified by users in customizing their own name spaces when constructing views of information to be shared with others.

A problem that arises from customized naming is the lack of name transparency: the same name may refer to different objects when used in different name spaces. To address this problem, naming mechanisms should support closure: each object should have an associated name space, and that name space should be used to resolve names that are specified by the object.

The goal of my work is to make it easier for users to cope with the increasing size of distributed systems. This goal has two parts: first, to make it easier for users to keep track of and organize those parts of the system in which they have indicated an interest; and second, to make it easier for users to learn about additional parts of the system which might be of interest, but which have not yet been seen by the user. Part of my thesis is that by fulfilling the first part, fulfillment of the second part will naturally follow: the structure imposed as users organize the information that is of interest to them is useful also to other users who have not yet seen that information.

In achieving this goal I developed a new model for naming that allows users to organize objects as they see fit. The model allows individuals, organizations (e.g., professional societies), libraries, and commercial services to organize information in many ways, and it allows users to choose among the alternatives, defining their own virtual systems (views of the system) based on the views defined by others.

The contributions of this dissertation include the recognition that users need the ability to view large systems in different ways and that a uniform global name space is not appropriate for user-level naming in extremely large systems; the recognition that systems that support multiple views or multiple name spaces must address the problems that result from the lack of name transparency; the presentation of the Virtual System Model, a new model for naming which better supports customization and provides powerful tools which help users organize the information available to them; and the design, implementation, deployment, and evaluation of Prospero, a prototype file system based on the Virtual System Model, which is being used to organize information available from Internet archive sites worldwide.

The Prospero prototype is used on more than 7,500 systems in 29 countries. Its rapid and widespread adoption demonstrates that the organizational flexibility provided by the Virtual System Model is useful. Prospero has been successfully used to organize information available from Internet archive sites including software releases, archives of Internet and Usenet mailing lists, and online copies of technical reports and conference papers. Individuals have used the prototype to organize information on many topics, and network service providers are starting to use it to organize information of interest to their users.

Prospero also has been adopted as the preferred method for remote access to archie, a database that maintains information about files available from key archive sites. It has also been adopted by the Australian Academic and Research Network (AARNet) as part of their plan for archiving in Australia [Cliffe 91]. AARNet has added Prospero support to one of their FTP servers where it helps them reduce the number of repeated file retrievals across a low-speed trans-Pacific connection, and it provides the benefits of Prospero to users who have not installed Prospero on their systems. A window based file browser built at the University of Rochester provides an alternate user interface to Prospero. Built on top of Prospero primarily to access the archie database, it is being extended to support browsing through other parts of the Prospero naming network.

1.2 Thesis Overview

Chapter 2 presents an introduction to naming. Terms that are used throughout this dissertation are defined. The role of names in computer systems is discussed and the manner in which names are shared and how they change are described. The chapter concludes by presenting a formal model for naming as it applies to existing systems, a model that will be extended in later chapters to provide better support for customization.

Chapter 3 continues the discussion by presenting an overview of naming in existing systems, highlighting the problems that arise as systems grow. For each problem, requirements are suggested that, if satisfied, help address the problem. The extent to which existing systems meet the requirements are described. These requirements form the basis for the Virtual System Model.

The Virtual System Model is described in Chapter 4, beginning with the definition of terms used when discussing the model. The key features of the model are then described informally. The chapter continues by formally describing the model and by discussing the differences between it and the model of naming presented in Chapter 2. The chapter concludes by discussing the relationship of the features of the model to related mechanisms found in existing systems.

Chapter 5 shows how the features of the Virtual System Model can be used to organize information in large systems. Users create customized views of the global system; this chapter describes the ways that users construct and add things to their own views and how they make use of other views. Some views are created specifically for use by others; the incentives to do so will be described. The Virtual System Model can be applied to all parts of a distributed system, and the relationship of the model to many of these parts is discussed.

The Prospero prototype is described in Chapter 6. The chapter begins by explaining what Prospero allows users to do. The structure of the prototype is then presented and its principal components are described. Prospero's support for some of the auxiliary functions related to file systems is then described. Among these functions are security, object mobility, garbage collection, and replication. The chapter concludes by presenting performance figures for the prototype. The Prospero user's manual and library and protocol specifications are included in several appendices.

Chapter 7 examines the use of the prototype in order to evaluate the acceptance and usefulness of the Virtual System Model. The chapter begins by presenting statistics which demonstrate the widespread use of the prototype. The information available through the prototype is described and the ways that users employ the prototype to organize and search for information are discussed.

The problem of finding and organizing information crosses several areas of research. While I have approached the problem from a systems perspective, paying particular attention to how people organize information, the problem can be approached from several other perspectives. In Chapter 8 I discuss the relationship of the Virtual System Model to alternative approaches and describe other work related to the problem.

Chapter 9 concludes by summarizing the contributions of this work. There is more to be done; possible directions are presented. The chapter closes with some final remarks.

Chapter 2

An Introduction to Naming

The problem of naming in computer systems has received a great deal of study. In this chapter I draw upon the existing work in the area to define some of the key concepts that will be required in the remainder of this dissertation. I discuss the nature of names, how they are used, and how they are created and evolve. I conclude the chapter by presenting a formal model for naming as it applies to the existing systems of interest in this dissertation. This model will be extended in later chapters to provide better support for customization.

2.1 Definitions

A *name* is the information used to identify an object. A name may be assigned by a user, or it may be a collection of information that uniquely identifies the object. It is usually a character string. The *binding* of a name is the object to which the name refers, though the object is often represented internally by an address.

By *object* I mean any part of a computer system, either physical or abstract, which can be uniquely identified. Examples of objects include files, directories, processors, users, and servers.

An *address* is the information used by a system to internally identify an object. An address is itself a name that is understood by the system [Saltzer 82]. As used in this dissertation, the distinction between a name and an address is that a name is visible to the user whereas an address might not be.

A *path name* is a name for an object that consists of more than one part. Each part is called a *component* of the name. Components are delimited by a special character called the *separator* which does not appear within any components. A name that consists of a single component is a *simple name.*

A *directory* is an object that maps a component of a name into an object. The object may itself be a directory. A *link* is an entry in a directory.

To *resolve* a name is to determine its binding. If the name has a single component, this may be accomplished by finding the mapping in the appropriate directory. If the name has multiple components, one must resolve the first component then resolve the remaining components of the name starting from the directory returned by resolving the first component.

A *naming network* is a directed graph often used to model the resolution of names. Its internal vertices represent directories and its edges are labeled with the initial components of the names that are resolvable from the directory represented by each vertex.

A *name space* represents a mapping of names to objects. The resolution of names in a name space can be modeled by a naming network together with a designated vertex (directory) from which name resolution is to begin. The designated vertex is called the *root* of the name space.

A *context* is one's state while resolving a name. It is the directory in which the current component of a name is to be resolved.

Closure is the mechanism that binds an object to a name space within which all names specified by that object are to be resolved.

Many of these terms will be defined in greater detail later in this chapter. Terms that apply only to the Virtual System Model are defined in Chapter 4.

2.2 The Nature of Naming

In her thesis, Sollins describes the way that names are used in everyday life [Sollins 85]. Among the characteristics she discusses, seven are particularly important in this work: individuality, multiplicity, mobility, locality, manifest meaning, communication, and sharing. These characteristics provide insight into the ways that humans are able to name the huge number of objects in the world, and that insight can be applied to the naming of objects in large computer systems. Additionally, by providing these characteristics in the mechanism used to name objects in computer systems, the naming mechanism will seem more natural to users.

By *individuality* it is meant that names are used differently by different people. Different individuals may use different names to refer to the same object, or they may use the same name to refer to different objects. Existing systems use various methods to allow users to customize the names they use. Examples include symbolic links and search paths. The advantages and disadvantages of these approaches are discussed in Chapter 3. Support for customization should be an integral part of a naming mechanism, not an afterthought.

By *multiplicity*, it is meant that an object can have more than one name. Multiplicity logically follows from individuality, but even a single individual may have more than one name for an object. The different names might be used in different situations. For example, a user might refer to a paper using its author's name if confronted with a set of papers on a particular topic. If confronted with a set of papers by a particular author, however, the user might refer to the paper by its topic or title. Of concern when supporting multiple names is the issue of whether all names are to have the same precedence, or if one name is primary, with all other names dependent on it.

By *mobility*, it is meant that the meanings of names change over time. For example, the name *annual-report* may refer to a different object in 1991 than it did in 1990. If users assign their own names to the annual-report (as individuality tells us they are likely to do) some may be referring to "the most recent annual report" while others are referring to the annual report for a particular year. The mechanism by which users keep track of objects should allow them to identify the objects to be remembered in either way.

Names are often used in a way that takes advantage of *locality*. The meaning of a name depends on the context within which the name is used. For example, the name *june* might refer to the host JUNE.CS.WASHINGTON.EDU when talking about disk usage, but it might refer to the month when talking about average rainfall. Naming mechanisms should make it easy to specify the context within which a name is to be resolved, and it should be easy to change the default context as the topic of interest changes. In most systems, directories provide the local naming context, and the working directory can be set according to the current topic of interest.

Though names refer to objects, they have a *manifest meaning* of their own. It is this manifest meaning that distinguishes names from addresses. The names assigned by humans convey information about the objects they identify. This meaning helps users remember the names of objects, and it helps users find the objects they are looking for. In systems where the names of objects have multiple components, each component of the name should have a meaning that is recognizable to potential users of the object. The manifest meaning of each component should reflect information that someone looking for the object is likely to know.

Names are used primarily for *communication*, whether between users, between programs, or even across time (names allow a user to remember an object that was seen at some point in the past). Names provide a shorthand for referring to objects. Communication requires that both parties be able to identify the object to which a name refers. The individuality of naming has the potential to make this more difficult.

Names are *shared*. Users don't assign their own names to each object they use. Instead, they frequently use names that have been assigned by others. The use of names assigned by others is one of the methods by which users identify objects of interest. Naming mechanisms should allow users to use names assigned by others, and they should provide ways for users to incorporate such names into their own name spaces.

Sollins's work concentrates on the mechanisms by which users jointly manage shared contexts. In her approach, shared names come to be accepted in stages and these stages are noted along with the name. I believe that the acceptance of a name is implicit in the number and nature of the users that use the name, and that it cannot be easily quantified. Users accept names by including them in their own views (name spaces). These names can be included either individually, or by adopting whole sets of names from the views of others. Users that don't like the names that are assigned either will pick other views on which to base their view, or will provide their own customizations, some of which might be adopted by other users. Acceptance is determined by the number of users to which a name is visible.

2.3 A Formal Basis for Naming

This section presents a formal model for the resolution of names in existing systems. This model is described informally in [Saltzer 78]. Comer and Peterson describe the model more formally and extend it to support the composition of names from different naming mechanisms (for example, /usr/bcn/readme@CS.WASHINGTON.EDU where /usr/bcn/readme is a filename parsed left to right and CS.WASHINGTON.EDU is a host name parsed right to left) and to model the iterative resolution of names, from those that users specify, to addresses, to the hosts on which the objects are stored, to the objects themselves [Comer & Peterson 89].

This dissertation is concerned with the names that are specified by and visible to users. The approach taken is to impose a uniform higher level naming mechanism that maps names from those specified by users to those used by existing systems. The naming mechanisms of the underlying systems are thus hidden from the user, and the names from the underlying systems should be thought of as addresses. Because this dissertation is not concerned with the resolution of these addresses, and because the naming mechanism visible to users is uniform, the extensions described in [Comer & Peterson 89] are not relevant to our discussion. The model presented in this section is a little simpler, but it does draw on the formalism presented in that paper. The model will be extended in later chapters to provide better support for customization.

A name can be broken into *components* delimited by a special character called a *separator*. Each continuous string of characters not containing the separator is a component. A *simple name* is a name that has only one component while a *path name* has more than one component.

A name space can be represented by a naming network together with a designated starting node in the network that represents the root of the name space. A naming network is a directed graph G $= (V, E)$ with labeled edges. Each vertex in V represents an object, possibly a directory. Each edge corresponds to an entry in a directory represented by the source of the edge, for the object represented by the destination of the edge. The edge's label is a simple name of the object relative to the directory that is the source of the edge. The labels are drawn from an alphabet Σ that contains all components of all names in the system, a potentially huge but finite set. An element of Σ is a component of a name, often multiple characters in length.

Figure 2.1 shows a sample naming network. When multiple name spaces share a common naming network it is the choice of the starting node that distinguishes name spaces from one another. Thus, the naming network in Figure 2.1 can be used to model more than one name space, including one rooted at q_1, one at q_2, and another at q_4.

Multiple naming networks may be represented as a single naming network with disconnected components. Thus, without loss of generality we may assume that there is a single naming network. This simplification allows us to completely specify a name space by identifying its starting node.

Presumably, the purpose of resolving a name is to identify an object so that it may be accessed. Thus, associated with each naming network we define an *access function*, $\alpha : V \rightarrow \mathit{type} \times \mathit{address}$. The access function maps vertices to the information needed to access the object represented by the vertex. This information consists of an address and information indicating how that address is to be used by the system.

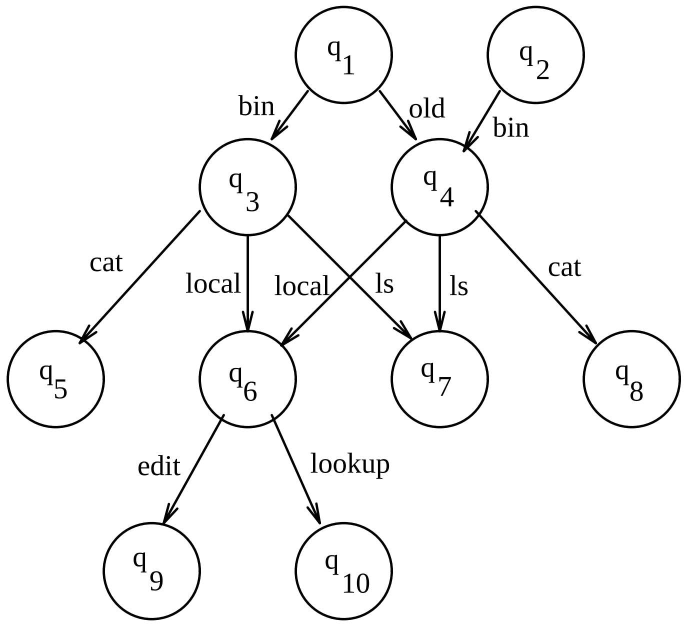


Figure 2.1: Sample naming network

To model the resolution of names we define a name resolution automaton, NA $= (Q, \Sigma, \delta, q_0, \alpha)$. Every vertex in the naming network is represented by a *context* $q \in Q$. The alphabet, Σ, is the set of labels on edges in the naming network. $\delta : Q \times \Sigma \rightarrow Q$, the *transition function*, maps a context and a component to a new context. Its definition is based on the set of edges in the naming network such that $\delta(q, \sigma)$ is defined as the destination of the edge in E from q with label σ, and is undefined if such an edge does not exist. q_0 is the initial context identifying the name space within which names are to be resolved. α is the access function defined for the naming network. Under the assumption that there exists a single naming network, Q, Σ, δ, and α will be fixed.

Naming automata are related to finite automata [Mealy 55, Moore 64, Hopcroft & Ullman 79]. The difference is that the result of applying a finite automaton is boolean, indicating whether the final context was accepting or non-accepting, whereas a naming automaton returns the result of applying the access function to the final context.

$$\begin{gathered} q = q_1 \\ q = \delta(q_1, bin) = q_3 \\ q = \delta(q_3, local) = q_6 \\ q = \delta(q_6, edit) = q_9 \\ return \quad \alpha(q_9) \end{gathered}$$

Figure 2.2: Resolving a name

We formally define the value of a directory as a function corresponding to the mappings in δ that define transitions from a particular context that represents the directory:

$$directory(c) = f(\sigma) : \forall_\sigma f(\sigma) = \delta(c, \sigma)$$

which means that the value of the directory corresponding to context c is a function such that for all values of σ the function maps σ to q if and only if the transition function maps context c and symbol σ to the new context q.

To resolve a name, an NA sets its current context to the initial context ($q = q_0$). It then reads a name from left to right (or from right to left for some naming mechanisms), and for each component it selects a new context by applying the transition function to the current context q and the current component σ ($q' = \delta(q, \sigma)$). When all components of the name have been read, the NA returns the address associated with the current context, $\alpha(q)$. If a component σ is read during name resolution for which $\delta(q, \sigma)$ is not defined, the NA returns a distinguished address indicating an error. Figure 2.2 shows the steps required to resolve the name "/bin/local/edit" using a naming automaton constructed from the naming network in Figure 2.1, and with starting state q_1.

Names are always resolved within a name space. Systems resolve names using a name resolution function which takes as arguments the name to be resolved and a name space (which identifies the starting context to be used by the NA when resolving full path names):

$$\text{resolve}(name, namespace)$$

The second argument (namespace) is not part of the name, nor is it specified by the user. Instead, the name space for the resolve function is filled in by the system. In many systems, the name space argument is constant, not changing from call to call. In others, it is determined by applying a closure function.

A *closure* function, $\kappa : object \rightarrow NS$, maps an object to the name space within which names embedded in or specified by the object are to be resolved [Saltzer 78, Neuman 89a]. Since the name space to be used is specified by the closure function, it needn't be specified as part of the name itself. Most systems approximate closure by associating a name space with the process resolving a name, and not specifically the object within which the name was embedded. Such systems keep track of a *working directory* for each process, and the working directory is used as the starting context. In some of these systems, processes can also define *search paths* and a name can be resolved starting from the successive elements of the search path until an object matching the name is found.

A *full path name* is a name that is to be resolved starting from the root of the active name space. Some systems allow names to be specified starting from a context other than the root of a name space. Such names, called *abbreviated names*, are syntactically different from full path names, and are resolved starting from an intermediate context that is remembered by the system. Technically, this can be thought of as resolving a name in a different name space, one whose root is the intermediate context, in which case the name would be a full path name. Traditionally, however, abbreviated names are treated separately, so I preserve the distinction.

Abbreviated names are sometimes called relative names, and full path names are sometimes called absolute names. I avoid this terminology because, in a system supporting multiple name spaces, even full path names are relative (to the root of the active name space).

Having presented a model for name resolution, I can now describe several properties about names and naming mechanisms in terms of the model.

If a full path name identifies the same object in every name space, then the name is said to be a *global* name.

$$\text{global}(name) \Leftrightarrow \forall_{ns_1,ns_2} \text{ resolve}(name, ns_1) = \text{resolve}(name, ns_2)$$

Names are *unique* within a name space if no two distinct names identify the same object.

$$\text{unique}(ns) \Leftrightarrow [\forall_{n_1,n_2} \text{ resolve}(n_1, ns) = \text{resolve}(n_2, ns) \Rightarrow n_1 = n_2]$$

A name space that includes non-unique names is said to support *synonyms*[1]. If a link is encountered during name resolution that yields a new name that must itself be resolved in the same name space, the link is called a *symbolic link* and a name that spans the link is called an *alias*. An alias is not a synonym; synonyms independently name the same object while the binding of an alias depends on the binding of the name returned when the alias is resolved.

A flat name space only supports single component names. This is the case when no transitions are defined from any of the nodes reachable from the root of the name space except for the root node itself. If the nodes reachable from the root of a name space form a tree with depth greater than 1, then the name space is hierarchical.

Two name spaces are said to be *disjoint* if they do not name any objects in common. The complement of disjoint is *overlap*; i.e., if an object is named in more than one name space, those name spaces are said to overlap.

$$\text{disjoint}(ns_1, ns_2) \Leftrightarrow \nexists_{n_1,n_2} \text{ resolve}(n_1, ns_1) = \text{resolve}(n_2, ns_2)$$
$$\text{overlap}(ns_1, ns_2) \Leftrightarrow \exists_{n_1,n_2} \text{ resolve}(n_1, ns1) = \text{resolve}(n_2, ns_2)$$
$$\text{disjoint}(ns_1, ns_2) \Leftrightarrow \neg \text{ overlap}(ns_1, ns_2)$$

[1]I do not consider abbreviated names to be synonyms.

2.4 Summary

In this chapter I discussed general concepts of naming. I started with a discussion of the ways that humans use names, with emphasis on those aspects that help in coping with the large number of nameable objects. I then presented a formal model for the naming mechanisms used in existing systems and defined some of the terms used to describe the mechanisms. This model will be extended in Chapter 4 to provide better support for customization and sharing.

Chapter 3

Existing Systems

This chapter continues the discussion of naming by presenting an overview of existing systems. It describes the file naming mechanisms used in centralized systems and discusses some of their shortcomings. It then describes several name servers, discusses the roles they play in the naming of files in distributed systems, and explores the advantages and disadvantages of various approaches.

As systems grow, it becomes less appropriate to support a uniform name space across all systems. The need for customization is discussed, and several systems that support customized name spaces are described. Customized name spaces introduce a number of problems which are discussed. The chapter concludes by suggesting requirements for naming mechanisms in large systems.

3.1 Centralized Systems

Early systems managed data by reading and writing directly to disk or tape. Later systems provided a file abstraction: programs could treat related data as belonging to a file, and the file could be accessed by name. Translation to the physical location at which the data was stored was handled automatically by the system.

As the number of files available within a system grew, the flat name space of early systems became cumbersome. Users had to agree how files were to be named so that they would not choose conflicting names. Additionally, the single catalog listing all files grew to the point that it no longer helped users determine the names of the files they were looking for; the listing was too long and too cluttered with the names of irrelevant files to be useful.

To address this problem, some systems supported separate catalogs for each user. While this allowed users to choose their own names for files without the possibility of conflict with names belonging to other users, it was not possible to share files. Only files within the user's catalog could be named, and thereby accessed.

This limitation is addressed in the Multics [Daley & Neumann 65] and Unix [Ritchie & Thompson 74] operating systems by supporting a hierarchical name space (an approach which has been adopted by most subsequent systems). Files are named within directories, and the *full path name* of a file or directory is the concatenation of the full path name of the containing directory and the name of the file within that directory. The directory structure forms a tree and names are resolved starting from a distinguished directory, called the *root*, which is known by all users.

3.2 Distributed Systems

Many of the advantages of hierarchical naming apply also to distributed systems. Name services such as the Internet Domain Naming System [Mockapetris 87], X.500 [CCITT 88], and DEC's Global Naming System [Lampson 85] use hierarchical names to reduce the need for centralized administration. Central administrators create directories that correspond to individual sites, and administrators at each site are able to administer names within their site's directory.

The distributed name services just mentioned are used to name hosts, users, servers, and even file systems, but not individual files. Despite this, these services play a role in finding files. The names maintained by these services are often used as prefixes of the names of files, and when they are, these services are used to identify the hosts on which the files are stored.

3.2.1 Host-Based Naming

Early distributed file systems employed host-based naming to identify files. Examples of host-based naming include IBIS [Tichy & Ruan 84], FTP, and to some extent, Sun's Network File System [Sandberg et al. 85][1]. In host-based naming, a user wanting to access a file must know the name of the host on which the file resides. While relatively simple to implement, host-based naming makes it difficult to organize and to locate information: the first part of a file name (the host) usually has little or no relation to the topic, and as a result, logically related information stored on different hosts ends up scattered across the name space. Additionally, when a file is moved, its name changes.

Because of these problems, users have adopted various ad-hoc solutions for organizing information. Some make local copies of information out of fear that it might move, or that they might forget where it is. Often the information is not used locally, but is copied "just in case" it is later needed. Others maintain lists of the information available from various locations, frequently updating the lists as things change. Some of these lists are periodically distributed through electronic mail.

3.2.2 Global Naming

An alternative to host-based naming is global naming. Global naming is used by the Andrew File System [Howard et al. 88], Coda [Satyanarayanan 90, Kistler & Satyanarayanan 91], Locus [Walker et al. 83], Unix United [Brownbridge et al. 82], Sprite [Ousterhout et al. 88], Echo [Hisgen et al. 89], and the Universal Directory Service [Lantz et al. 85]. In these systems all file names are part of a single name space and the name of the system on which a file resides is not explicitly part of the file's name.

[1]In NFS, the user must know the name of the system on which the file is stored in order to mount the file system. Once mounted, however, the host name need not be part of the file's name.

The Universal Directory Service uses the same name space for all naming, not just the naming of files.

These systems address, in part, some of the problems of host-based naming. In particular, the name of the storage site is no longer part of a file's name. As most of these systems are implemented, however, files are distributed across servers based on prefixes of their names. This reduces the size of the naming database and reduces the number of requests to name servers, but it also constrains the distribution of files in such a way that files whose names share a common prefix, i.e. those that are "related" in the name space, must usually be stored on the same storage site.

3.3 Problems with Existing Approaches

Hierarchical naming has worked well in existing systems, but as implemented, it has several limitations. Further, as systems grow, other problems arise which are inherent in the approach itself. This section describes problems of both kinds.

3.3.1 Synonyms

The file systems described so far provide only partial support for synonyms: in Multics, an object can only have multiple names within a single directory; in Unix, only within a "file system" (a boundary which is supposed to be hidden from the user). Existing distributed file systems do not support synonyms across system boundaries.

Lack of true synonyms creates problems when a file should logically appear in more than one directory. Many of the systems described so far support aliases (symbolic links), but when an alias is resolved, a new name is returned which must be further resolved. The problem with aliases is that they depend on a primary name for the file they reference. If the primary name changes, or is removed, the alias ceases to work.

3.3.2 Finding Things

Large name spaces are often easier to administer when they are organized hierarchically. Sub-hierarchies are assigned to users and organizations, and those users and organizations are responsible for assigning names within their part of the name space. Unfortunately, this results in a name space whose top levels are often the names of organizations, and whose second levels are the names of users. This means that logically related information ends up scattered across the name space. As was the case for the storage site, users searching for information on a particular topic will not often know the name of the user or organization that maintains the information.

An approach to solving this problem is to create shared directories with links to information on particular topics. The difficulty with this approach is that, in a large system, people will not agree on what information is important. Instead it becomes necessary to decide which shared directories should exist, and for each directory, to assign the task of maintaining it. Once a system crosses administrative boundaries, gaining consensus becomes even more difficult.

A uniform hierarchical name space works best for systems where a single administrator can make decisions about what should appear in the upper levels of the name space. Users can often accept the role of a central authority in assigning unique organization names, for example domain names, but when it comes to matters of opinion, e.g., what topics deserve shared directories and who should maintain them, users are less willing to accept someone else's judgment, at least not if they don't personally get to decide whose judgment to accept.

This problem is very apparent on Usenet, a distributed message service for disseminating messages on many topics, worldwide. A large proportion of the messages sent on Usenet discuss what messages are appropriate for particular newsgroups, whether new newsgroups should be created, and what they should be called. This clearly demonstrates the problem of reaching consensus on globally shared names.

3.3.3 Customization

A second shortcoming of the uniform hierarchical name space is poor support for customization: all users see the same name space. Customization is important for several reasons. In large systems, there is a huge amount of information, and much of that information is not of interest to a particular user. Customization helps the user to remember, and later return to, information that is of interest. Examples of customization can be seen in systems that allow users to define alternative names for frequently referenced files.

But customization has many other uses. Different users employ the same names to reference different files. For example, if two similar files are stored in different parts of a system, users in different parts of the system might use a common name to refer to the particular file that is nearer to them. Customization is also used when different versions of a file are needed in different situations. For example, different system binaries are needed when running on different architectures, but it should be possible for users to use the same name for a program or file regardless of the type of system they are using. Customization is also needed when users want to change the behavior of the system. They customize their name space so that their own modified version of a file or program is used instead of the one provided by the system. Finally, customization is important because users don't want to type long path names whenever they refer to a file. Long path names are often necessary to uniquely identify a file, but in a customized name space, shorter, or abbreviated, names can be used for those files that are frequently used.

Existing systems support customization and abbreviated names through mechanisms external to the name space. For example, TOPS-20 [Dig 80] and VMS [Dig 88] allow users to use logical names in place of file names. Once specified, logical names support limited customization of the name space, but they are private to the process that specified them. R* [Lindsay 81] supports private aliases which allow users to assign their own names to remote databases. Most systems allow users and programs to specify search paths to be followed when resolving abbreviated names; the system tries each directory in the search path until a match is found. The Andrew File System supports both a local and a global part of the file system. The local part of the file system is customized on a system-by-system basis, allowing each system to be configured in an appropriate manner. Most of the file names specified by users, however, fall into the uniform, global part of the name space.

The problem with the customization mechanisms described so far is that the customizations are not an integral part of the naming system. They are visible only to the user who has made the customization. This makes it difficult for users to resolve customized names that were specified by other users. For example, compilers often use search paths to identify the directories to be searched when looking for libraries. These search paths are found in environment variables set by the user. When a user attempts to compile a program that was written by another user, it might not be possible to resolve some of the names if the search paths of the two users are different.

In many systems, users can include symbolic links that assign short names to the files that they frequently access. However, most of the systems described so far allow users to assign only names that appear in their own sub-hierarchies of the file system. This is not a sufficient level of customization. Users should be able to customize all parts of their name spaces. Without this ability, users must remember when they have created customized views of parts of the name space that they are not allowed to customize directly. Finally, users who expect others to be able to resolve their abbreviated names must first prepend the names of their home directories, thus turning the abbreviated names back into full path names.

3.4 User-Centered Naming

One reason it is hard to find things in large systems is that there is a huge amount of information, and much of that information is not of interest to a particular user. This problem is addressed in Tilde [Droms 86, Comer et al. 90], QuickSilver [Cabrera &

Wyllie 88], Plan 9 [Presotto et al. 91], and Amoeba [Tanenbaum et al. 90, van Renesse 89] by supporting *user-centered* naming: each user resolves names in a separate name space containing only the files of interest to that user.

The customization supported by these systems is important for a number of reasons: it reduces clutter in the user's name space; it allows users to define shorter names for frequently referenced files; and it allows users to replace entire portions of the name space with alternative views that are more appropriate for their needs. A user-centered name space also eliminates the need for consensus when deciding what should appear in the upper levels of the name space. Each user can make that decision based on his or her own opinions.

User-centered naming presents several problems of its own. The first problem is the lack of name transparency: the same name might refer to different files when used in different name spaces. This has the potential to make sharing difficult and it can cause confusion. As was the case with earlier customization mechanisms, part of this problem results from the fact that in most of these systems, customizations are visible only to the user or process that made them.

Another problem with user-centered naming as supported by Tilde, QuickSilver, and Plan 9 is that a file (or collection of files) must be explicitly added to the name space before it can be accessed[2]. This requires that the user specify a globally unique name for the file, thus reintroducing all of the problems associated with global naming.

A final problem with user-centered naming as supported in Tilde, QuickSilver, Plan 9, and Amoeba is that the information that is available changes, and it isn't practical to expect users to keep their name spaces up to date, item by item. There needs to be some way to specify a view of information as a function of one or more other views. Existing systems do not provide adequate tools for constructing derivative views. In Tilde and Plan 9, part of the problem is that views are not persistent. Instead, they are

[2]Naming in these systems might be better described as *user-exclusive* rather than *user-centered*, since files (or collections of files) that have not been explicitly included in the name space cannot be accessed.

constructed by a process (often using a configuration file) and they only live as long as the process that constructed them.

3.5 Summary

This chapter described the naming mechanisms used by a representative set of existing systems and discussed their limitations. It identified some of the problems that arise from supporting a uniform global name space in systems that cross administrative boundaries. The need for customization was presented, and the customization mechanisms employed by traditional systems were described. Several recent systems support user- or process-centered name spaces. These systems provide better support for customization and address some of the problems inherent with the uniform global name space, but they run into problems due to their lack of name transparency. Table 3.1 shows the systems that were discussed in this chapter and lists some of their characteristics.

The amount of information that is available within distributed systems has been growing. To help users organize and share this information, naming mechanisms should allow objects to have multiple names. These names should be synonyms. If one name changes, the others should continue to work. Naming mechanisms should support customization, but it should always be possible to correctly resolve a name, even if the name was specified by a user in a different name space, or by a user that has applied different customizations. Finally, tools are needed to help users define views of files as functions of one or more other views. Users should not be required to build their name space item-by-item. These requirements form the basis for the Virtual System Model.

Table 3.1: The characteristics of existing naming mechanisms

	Characteristics				
System [a]	Scale	Objects	Namespace	Synonyms	Customization mechanisms
Multics	system	files	uniform	aliases[b]	search paths
Unix	system	files	uniform	aliases[c]	search paths
DEC Global Naming	global	filesystems	uniform	aliases	working root
IDNS	global	hosts	uniform	aliases	resolver can support search paths
X.500	global	hosts/users	uniform	aliases	context of resolver
FTP	WAN	files	uniform	no	none
NFS	LAN	files	system-centered	aliases	mount
AFS	global	files	uniform[d]	aliases	search-paths
Coda	global	files	uniform	aliases	search-paths
Locus	LAN	files	uniform	aliases	search-paths
Sprite	LAN	files	uniform	aliases	search-paths
Amoeba	WAN	objects	user-centered	yes	per-user root
Plan 9	company	files	per-process	no	mount, union-mount
Quicksilver	company	files	user-centered	yes[e]	links, search-paths
Tilde	LAN	files	per-process	of trees	per process naming of trees
Prospero	global	files	object-centered	yes	closure, filters, union-links

[a] Grouped by predominant characteristics
[b] Synonyms supported within same directory
[c] Synonyms supported for files on same file-system
[d] For the /afs hierarchy
[e] But if across system boundaries, only for immutable files

Chapter 4

The Virtual System Model

This chapter presents the Virtual System Model, a new model for organizing large systems. The Virtual System Model has four principal features: support for customizable name spaces, tools to help users construct name spaces, support for synonyms, and a method for selecting the appropriate name space when resolving a name. These features satisfy the requirements that were summarized at the end of Chapter 3.

The chapter begins by defining terms that apply to the Virtual System Model and by discussing the features of the model informally. The chapter continues by formally describing the Virtual System Model and by discussing its differences from the naming model presented in Chapter 2. The relationship of the features of the Virtual System Model to the mechanisms found in existing systems is then described.

Although the examples in this chapter concentrate on the naming of files, the Virtual System Model also applies to other aspects of a distributed system. Chapter 5 will describe this.

4.1 Definitions

When used in this dissertation, the term *system* usually refers to a distributed system: a collection of computers, connected by a computer network, working together to implement some set of services. In some cases, if the meaning is clear, the term also refers to the *local system*, the physical system to which a user is logged in, on which processes execute, or on which files are stored.

A *virtual system* is a system that is composed of files, processors, services, applications, users and other objects available over a network. A virtual system is created by identifying the objects of interest and specifying the names that will refer to those objects from within the virtual system. This mapping of names to objects constitutes the virtual system's name space; the name space plus the objects it names constitute the virtual system.

The term *view* is a general term describing a mapping from names to objects. It is often qualified by describing the objects to which the view applies or the mappings from which it is derived. For example, a name space represents a view of all the objects in a system and consists of the complete set of mappings of names to objects that apply during a particular call to the name resolution function (for all possible names to be resolved). Similarly, a directory defines a view of the objects in a name space whose names share a common complete prefix[1]. An alternate view of a directory might define a different set of mappings to the same objects. Because it includes a name space, a virtual system imposes a view on the objects it contains.

A *conventional link* is similar to a hard link in traditional file systems: it maps a name of an object to the information needed to access that object.

A *union link* is a link to a directory that causes the links that are part of the linked directory to appear as part of the directory containing the union link. A directory's

[1]By complete prefix I mean all but the last component of the name. An alternate description of a directory is that it defines a view of the objects reachable in a single step from a particular node in the naming network.

contents are the union of the set of conventional links it contains and the contents of all directories included through union links.

A *filter* is a program attached to a link. A filter changes the results of queries to the directory that is the target of the link when reached through that link.

A *link* is either a conventional link or a union link, with or without an attached filter.

A *virtual directory* is a directory in a virtual file system. The contents of a virtual directory might be calculated at the time the directory is queried by applying filters or expanding union links.

4.2 Features of the Model

There are four principal features of the Virtual System Model. These features are support for a customizable name space, tools to help the user define his or her name space, support for synonyms, and a method for selecting the appropriate name space when resolving a name. This section shows how these features are supported by the Virtual System Model and how they satisfy the requirements set forth in the previous chapter.

4.2.1 Customization

The Virtual System Model supports customization by providing multiple name spaces. This allows users to organize information in the manner they desire, making it easier for them to keep track of the information they have seen. Users are not restricted to organizing objects that they own; they can just as easily impose their own organization on files and directories owned or maintained by others. When organizing information it is possible to create a customized view of the objects in a directory that is specified as a function of the original directory, or of another view. This is accomplished using a filter or a union link, each of which is described later.

To create a new name space (a customized view of available objects) a user creates a directory that is to be the root of the name space[2]. Subdirectories are then created

[2]A new directory is required since existing directories must remain unchanged for use by those that don't want to see the customization.

and links to existing objects or directories of interest are added with names that are meaningful to the user. If the organization of a piece of an existing name space is acceptable to the user, a link to that piece can be added, and the piece becomes part of the user's name space. When a user wishes to include part of an existing name space, but with changes, a link to the desired part can be added, and filters and union links can be applied to change the way it appears in the new name space.

4.2.2 Filters

A filter is a program, attached to a link, that allows the view of the target directory to be altered. The input to a filter is the value of the directory that is the target of the link. The value of a directory is a list of the links it contains. Each link consists of a simple name for the link, a target of the link, a link type, and if the target of the link is itself a directory, optionally a filter to be applied when that directory is listed or used to resolve subsequent components of a name. A filter returns a list of links (the value of a virtual directory) that results from applying the filter to the input. It is the list of links returned by the filter that is displayed when a user lists a virtual directory or uses the virtual directory to look-up a component of a name.

What a filter can do

A filter can remove links from the list of links read from a directory, add new links, change the simple name of a link already in the list, change the target of a link, change the type of a link, change the filter associated with a link, or add new filters to a link. A filter operates on a copy of the links from a directory and its only effect is on the list of links it returns when the directory is queried through the filter (i.e. when the directory is reached by the link to which the filter is attached). A filter cannot have side effects[3]. Every link returned by the filter must have as a target a node in the naming network.

[3]Side effects can be useful for some purposes and might be allowed by particular implementations, but side effects are not supported by the model.

Although a filter can only affect the list of links it returns, it is free to use any information it can obtain even if that information is not part of the naming network. For example, a filter might query a separate database and return the results of the query as a virtual directory. A filter is free to ignore its input; this is equivalent to deleting all the links from the input and adding links of the filter's own choosing. Even in this situation, the filter only has an effect through the list of links it returns.

Limitations

The application of a filter is specified by associating the filter with a link, and as such it is not possible to maintain multiple links to the result of the application of a filter. It is possible to create an additional link that applies the same filter, but the additional link would not be affected if the filter associated with the original link were changed. This is undesirable because when a user finds a directory of interest, it should be possible to treat the directory the same way regardless of whether the directory corresponds to a node in the naming network, or the result of applying a filter. It is possible to work around this limitation by attaching a filter to a union link. The union link is described in Subsection 4.2.3.

Because a filter's only effect is on its return value, and since its return value is a single directory, a filter can not directly affect the view of any directory except that to which it has been applied. However, by changing the filters associated with the links it returns or by adding new filters, a filter can indirectly affect subdirectories. This mechanism will be described in greater detail shortly.

Example of a simple filter

Figure 4.1 shows an example of the results of applying a simple filter. The filter in the figure removes links that do not match the wildcarded string `*.ps`. It is applied to a directory that contains several files, but the computed view shows only those files that have the appropriate suffix, indicating that they are papers in PostScript format. In

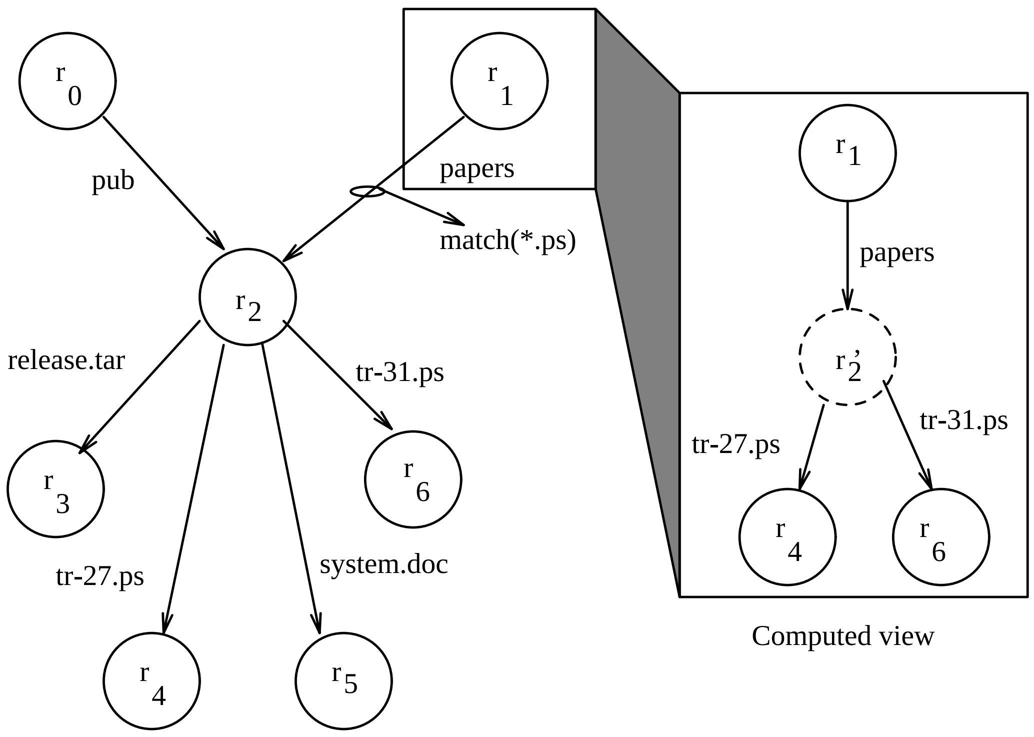


Figure 4.1: Results of applying a simple filter

the figures in this section, solid circles represent physical files or directories. The label inside the circle exists only for ease of reference in the text of this dissertation; the label is **not** part of the name of the file or directory as seen by the user. The directory r_2' in Figure 4.1 does not exist as a node in the naming network. Instead, what is shown in the computed view is the result of applying the filter to directory r_2.

This example uses a parameterized filter. The parameter, `*.ps`, specifies the names of the links to be matched. The parameter allows a single filter program to perform multiple filtering operations. When applied with different parameters, a filter program should be thought of as a different filter. It performs a different filtering operation, e.g., matching names ending in `ps` rather than names ending in `txt`. The parameters of a filter are set when the filter is attached to a link.

Composition of filters

A link may have multiple filters attached. The order of application of the filters is important. For example, if one filter removes all links whose names contain the letter `A` and a second filter renames links by changing all `B`s to `A`s, then one ordering will yield the renamed links while the other ordering will not. Neither ordering is intrinsically correct; it is up to the user to specify the correct order when the filters are attached.

Throughout most of this chapter I make the simplifying assumption that a link will have at most a single filter attached. This assumption can be made without loss of generality since the composition of two or more filters is itself a filter. Thus, where more than one filter is needed, the filters can be represented by a single filter, the result of composing the necessary filters in the appropriate order.

How a filter affects more than a single directory

Although a filter can only directly affect the directory to which it is attached, it can indirectly affect subdirectories by changing the filters to be applied when subdirectories are listed or used to resolve subsequent components of a name. This works because the filters to be applied to subdirectories are specified by links in the current directory.

This ability to indirectly affect subdirectories can be used to apply a filtering operation to a directory hierarchy. For example, to filter out all files more than a month old from a hierarchy, a filter scans the links from its input (the value of the directory to which it is attached). For each link it checks the type. If the link is to a file, then the filter checks the file's write date, and if more than a month old, it removes the link. If the link is to a directory, then the filter attaches itself to that particular link. If a filter is already associated with the link, then the current filter (the one being applied) attaches itself in composition with the existing filter, in this case composed so that the existing filter is applied first. When a subdirectory is listed or used to resolve a name, one finds that the subdirectory corresponds to exactly the same node in the naming network as if the filter had not been applied to the current directory, but that the very same filter applied to the current directory is now applied to the subdirectory as well. The filter will continue to propagate itself downward through all descendant directories in the same manner.

When a filter changes the filters applied to subdirectories it is not constrained to leave existing filters in place, nor is it otherwise restricted in the choice of the filters to be attached. Although the code that implements the filters to be attached must already exist, new filters can be specified by composing existing filters or by selecting new parameters for parameterized filters. The example in Section 4.3 (Figure 4.6) shows a filter that creates new filters which are then attached to subdirectories. In this case the **distribute** filter attaches the **attribute** filter to subdirectories and the parameters for the **attribute** filter are constructed from the author attributes of the files in the current directory.

Changing the filters attached to links returned also allows the creation of *ghost* hierarchies, parts of the name space that appear to be specified entirely within the filter. Although the ghost hierarchy appears to be specified entirely within the filter, each directory in the ghost hierarchy is associated with a node in the naming network, usually a node reachable from the node to which the original filter was applied. Multiple directories from the ghost hierarchy might share the same node from the naming network, but the filters attached to links change the way each directory appears.

4.2.3 Union Links

A union link allows links from another directory, one corresponding directly to a node in the naming network or one resulting from the application of a filter, to appear as if they originate from the node in the naming network from which the union link originates. When a union link is present, the contents of the target of the union link (which must be a directory) appear to be included in the originating directory[4]. If a union link has a filter

[4]If the target directory contains union links, those links are also expanded. The result is the union of the conventional links in the transitive closure of the original directory under the union link relation.

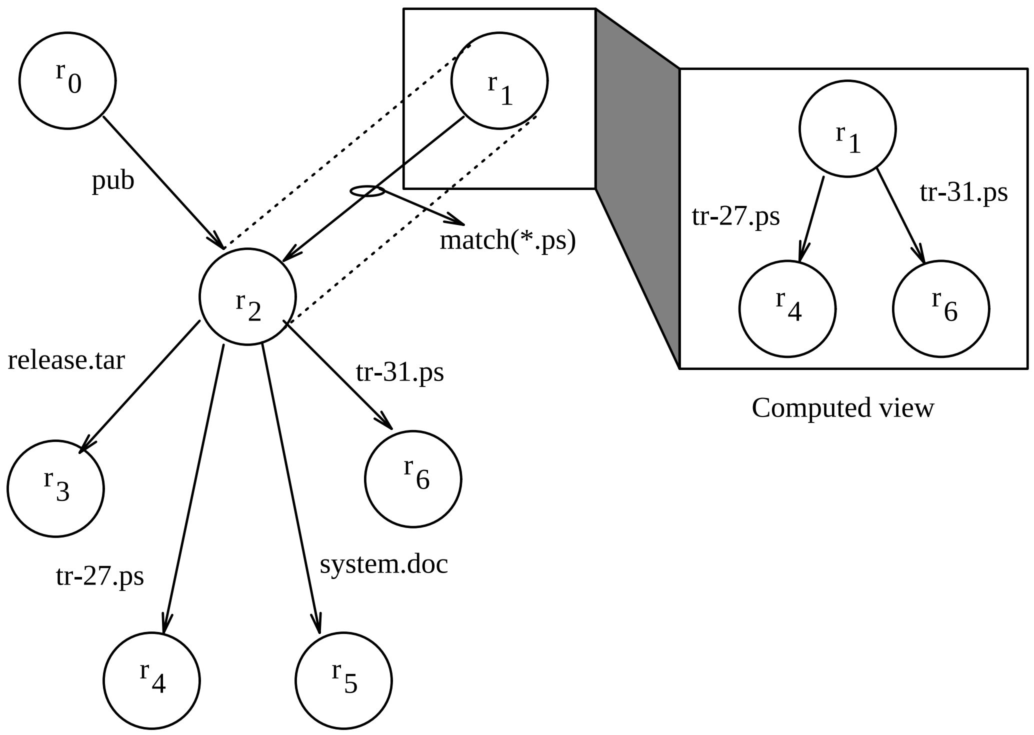


Figure 4.2: Use of a filtered union link

applied, then the links that are included are those returned when the filter is applied. When traversed while resolving a name, a union link does not consume a component of the name. As such, in the naming network, an edge corresponding to a union link will not be labeled with a component of a name[5].

Figure 4.2 shows the results of applying the filter shown in Figure 4.1, but this time the filter is attached to a union link. Directory r_2' in the first example was a computed view and could only be referenced through the link `papers`. In this example, directory r_2' from the previous figure has been included in the computed view of directory r_1, and links can be made to r_1 the same way they can be made to any other node in the naming network. If at a later date, the filter attached to the union link were to change, the change would be visible to all users with links to r_1.

[5]The edge might be labeled with a filter if one is to be applied.

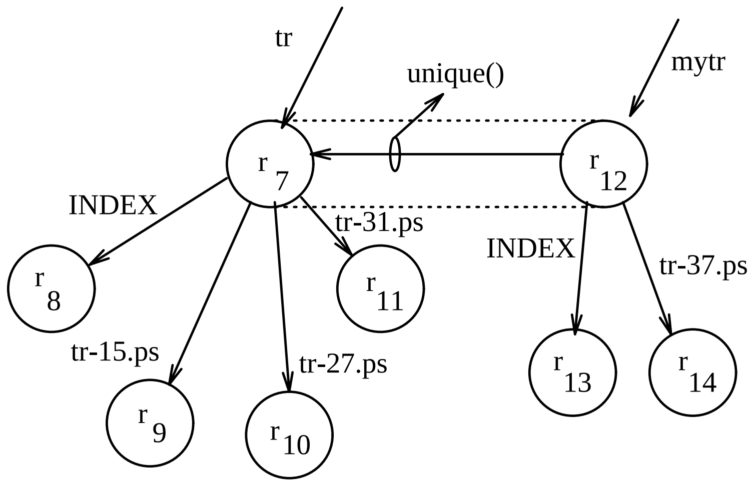


Figure 4.3: Use of a union link for customization

Union links have a second function: when a union link is combined with a filter that eliminates duplicates[6], the user is able to create a customized view of a directory by creating an empty directory, adding a union link to the original directory, and making changes to the new directory.

For example, in Figure 4.3, directory r_{12} presents a customized view of directory r_7. The link `tr-37.ps` has been added to the customized view and the link `INDEX` has been modified to reference object r_{13} (instead of object r_8). These changes are only visible when looking at r_{12}. They are invisible to anyone looking directly at r_7. Future changes to directory r_7 will be visible to individuals looking at r_{12} as long as the changes do not conflict with customizations already made to r_{12}.

If the **unique** filter is applied to a union link, the order in which union links are added to a directory is significant. If more than one of the included directories has a link with a particular name, the link from the directory searched first is the one returned. The local directory is searched first, then directories are searched in the order that the corresponding union links were added to the local directory. This behavior is analogous to that of search paths in existing systems.

[6]This filter is called the **unique** filter. In the prototype implementation it is implemented as part of the union link mechanism itself. This greatly improves its efficiency.

Filters and union links are powerful mechanisms for supporting customization and the manipulation of name spaces. Filters are written in standard programming languages and can take any action that can be specified in such languages. The union link allows the manipulation performed by a filter to affect the directory containing the filter.

4.2.4 Synonyms

The Virtual System Model allows links to be made from any directory to any object. Unless the creator of a link specifically indicates otherwise, links are synonyms: they are additional names for the referenced object and they will continue to work even if other names for the object change. An object will continue to exist until all links to it have been removed. The ability to add links to objects is not constrained by system boundaries or by the type of the target of the link. It is possible for links to appear from any directory to any object without regard for the physical location of the two objects.

Although names are synonyms and do not depend on the existence of other user-level names, implementations may represent objects by system-level names. System-level names might be human-readable, but they should be thought of as addresses. If an implementation represents a link as a mapping to a system-level name, and if the system-level name changes, it is the responsibility of the implementation to make sure that all links continue to work.

4.2.5 Closure

So far, this section has shown how the Virtual System Model can support multiple name spaces and it has described the tools that help users construct customized views of existing name spaces. Support for multiple name spaces leads to problems with name transparency: the same name might refer to different objects when used from different name spaces. This can make a system confusing and can hinder sharing.

This problem is addressed in the Virtual System Model by closure: every object that is created using the Virtual System Model has an associated name space. The name space is represented as a reference to the node in the naming network that is the root of the name space, and the reference is stored in the attribute list for the object. Usually the name space *closed* with an object (associated with the object by closure) is the name space of the process that created the object, but the closed name space may be specifically set by the object's owner.

When a name is resolved, the name is resolved in the name space closed with object that specifies the name: if a name is specified within a program, the name space closed with the program is used; if specified by the user as an argument to the program, the user's name space is used; if read from a data file, the name space closed with the data file is used. Through closure, the context within which a name is to be resolved is automatically determined. Although the same name may refer to different objects within different contexts, the correct context is always known.

4.2.6 Review of the Features

This section presented the features of the Virtual System Model. The naming network supported by the Virtual System Model forms a *generalized directed graph*, a directed graph that is augmented by filters and union links. The features of the Virtual System Model satisfy the requirements set forth in Chapter 3. Customization is supported by allowing multiple starting nodes, each corresponding to a separate name space. Filters and union links make it easier for users to construct and maintain customized name spaces by allowing them to define views of directories as functions of one or more other views. Because the directed graph is not constrained to form a tree, or even to be acyclic, synonyms are supported for all objects. Finally, closure specifies the name space within which names are to be resolved, making it possible to correctly resolve a name even if the name was specified by a user in a different name space.

4.3 A Formal Presentation

This section presents a formal description of the resolution of names using the Virtual System Model, based on the model for name resolution presented in Section 2.3, extended to provide better support for customization.

4.3.1 The Resolution of Names

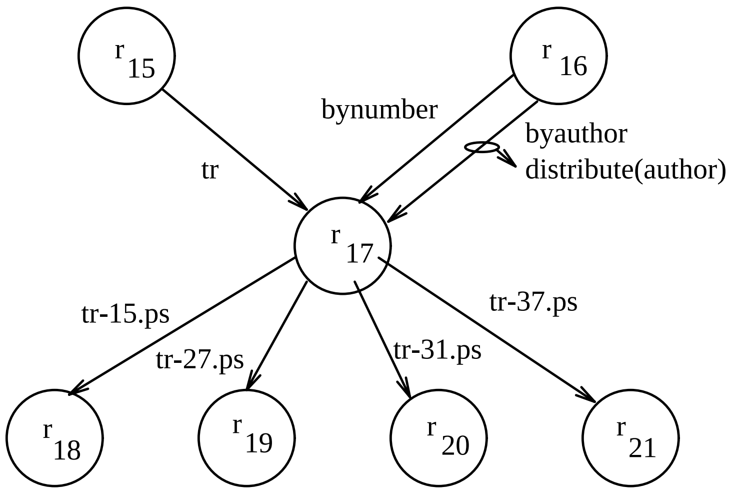
As was the case for existing systems, the resolution of names in the Virtual System Model can be modeled by a name resolution automaton, NA $= (Q, \Sigma, \delta, q_0, \alpha)$. Every nameable object, whether real or computed, is represented by a *context* $q \in Q$, and q_0 is the initial context. The alphabet, Σ, is a set containing every component σ that appears in any valid name. $\delta : Q \times \Sigma \rightarrow Q$, the *transition function*, maps a context and a component to a new context. $\alpha : Q \rightarrow \mathit{type} \times \mathit{address}$, the *access function*, maps contexts to the information needed to access the object represented by the context. $Q, \Sigma, \delta, \alpha$ are defined by the naming network and are thus fixed under our single naming network assumption. A name space is identified by specifying q_0.

Filters

Unlike traditional systems, in the Virtual System Model a context for name resolution q consists of a real part r corresponding to a node in the naming network, and a function[7] f that is to be applied to the real part, that is $q \in R \times F$. The transition function maps a context and a component of a name to a new context. In so doing, it applies the function part of the context to the mappings in the directory that corresponds to the real part. It then looks up the component and returns the new context. The new context itself consists of a real part and a function. For most contexts, the function will be the identity function.

Graphically, the naming network is still represented by a directed graph with labeled edges and vertices. Each vertex corresponds to the distinct real parts of the contexts in

[7]As discussed earlier, the function might be the composition of two or more functions.



Object	Value of author attribute
r_{18}	neuman
r_{19}	lazowska
r_{20}	lazowska
r_{21}	neuman

Figure 4.4: Sample naming network for the Virtual System Model

Q. Instead of being labeled with only a component of a name, the edges in the naming network are also labeled with the function that is to be applied to the real part of the next context (the destination of the edge). To avoid clutter, the function will be left out when it is the identity function. When moving through the naming network while resolving names, it is necessary to remember the function associated with the last edge over which one reached a vertex. The function and the vertex together specify the context. Figure 4.4 shows a sample naming network for the Virtual System Model. The table in the figure lists values of the author attribute stored in the attribute list of four of the files. These attributes are read by the **distribute** filter[8].

It is important to note that the Virtual System Model only supports the resolution of names in the forward direction. In particular, it only supports the resolution of full path names or of abbreviated names starting from a context that has previously been resolved

[8]As an arbitrary program, a filter can query the attributes of an object in much the same way that it reads or otherwise accesses the object.

$$\begin{gathered}
q = [r_{16}, I] \\
q = \delta([r_{16}, I], byauthor) = [r_{17}, \mathbf{distribute}(author)] \\
q = \delta([r_{17}, \mathbf{distribute}(author)], neuman) = [r_{17}, \mathbf{attribute}(author = neuman)] \\
q = \delta([r_{17}, \mathbf{attribute}(author = neuman)], tr \Leftrightarrow 37.ps) = [r_{21}, I] \\
return \quad \alpha(r_{21})
\end{gathered}$$

Figure 4.5: Steps to resolve a name in sample naming network

and saved (e.g., a working directory). The Virtual System Model does not directly support the resolution of relative names of the form "go back two directories, then resolve name1/name2" as might be specified by the Unix path name `../../name1/name2`[9]. In the prototype implementation of the Virtual System Model, the user interface interprets such paths and performs a textual substitution before the name is resolved.

Figure 4.5 shows the steps required to resolve the name `/byauthor/neuman/tr-37.ps` using the naming network in Figure 4.4 starting from node r_{16}. The box labeled "Computed view" in Figure 4.6 shows the result of applying **distribute**(*author*) to r_{17}. The result is a set of new links corresponding to the unique author attributes in the files under r_{17}. The target of each of the links is r_{17} itself, but each link has a different filter applied, one that will pass only those links whose targets possess the appropriate author attribute. In Figure 4.5 note that the access function is applied to the real part of the final context. If the final context has a functional component other than the identity function, the function will also be returned by the name resolution function[10].

Figure 4.7 shows how the name space rooted at r_{16} appears to the user. Although Figure 4.7 more directly reflects what the user sees than Figure 4.4, it is unable to model the effect of change. In particular, it is not possible to determine from Figure 4.7 how r'_{17}, r''_{17}, and r'''_{17} will be affected when r_{17} changes.

[9]Unix does not really support such names. Instead, each directory has an entry named ".." that refers to its parent directory. Thus, a name of the form ../../name1/name2 is really just an abbreviated name starting from the current working directory. The ".." notation in Unix appears to allow the user to back up because Unix directories can only have a single parent. Symbolic links violate this assumption, and the ".." notation in Unix can give unexpected results when such links are crossed.

[10]Such a function might implement an alternative method to access the object.

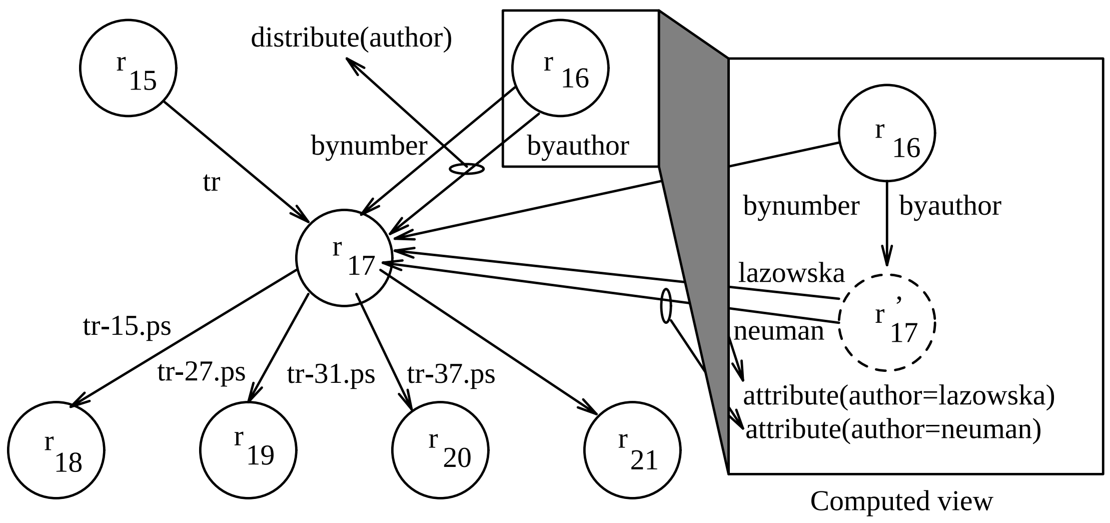


Figure 4.6: Result of applying **distribute**(*author*) to *r17*

Union Links

So far my discussion of the differences between existing naming models and the Virtual System Model has concentrated on filters. A second difference is the inclusion of union links. A union link is represented by a transition that consumes zero components of a path name. If, after any filters have been applied, a union link appears in a directory, then in addition to returning any links in the current directory that match the component, the transition function will recursively call itself on any contexts referenced by union links. If a vertex is reached that has already been visited during resolution of the current component, the context will not be expanded further[11]. The transition function might return multiple matches[12]. Thus, the transition function must be extended so that it can return multiple contexts, $p \subseteq Q$. It must also be extended to accept multiple contexts for input, returning the union of the results of resolving the component in each of the contexts.

[11]In a well-formed naming network (see Subsection 4.3.2) such pruning does not affect the correctness of the expansion.

[12]If the **unique** filter is applied to a union link, it will only be expanded if no match for the component of the name has already been found.

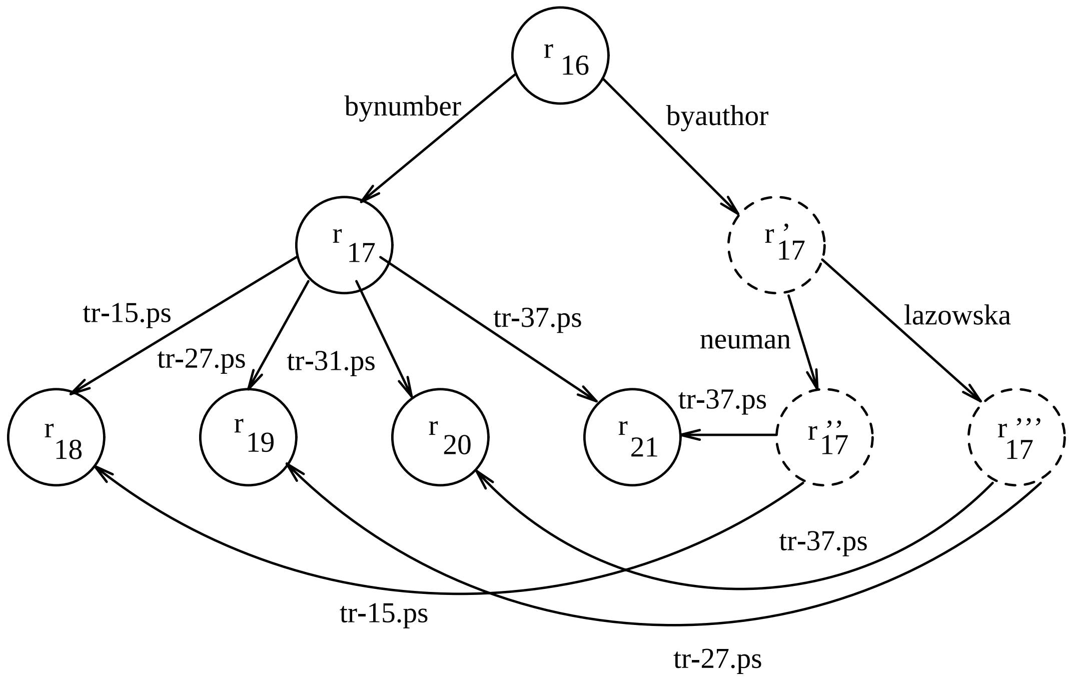


Figure 4.7: The way the user sees the naming network

The difference between the transition function in existing models of naming and that required to support union links is precisely the difference between a deterministic finite automaton (DFA) and a non-deterministic finite automaton (NFA). The NFA is described in [Rabin & Scott 59] and [Hopcroft & Ullman 79] where it is shown to be computationally equivalent to the DFA.

Graphically, a union link is represented as an edge with no label [13]. Figure 4.8 shows a naming network with three union links. Node r_{22} has more than one union link, and the numbers at the base of the links indicate the ordering of the links in the directory.

To resolve the name `tr-37.ps` in the naming network of Figure 4.8, directory r_{22} is checked for the component `tr-37.ps` by applying the transition function. After checking for local links, the transition function recursively calls itself for the directories referenced by each of the union links, in order. It finds a link labeled `tr-37.ps` under directory r_{31},

[13]The edge might be labeled with a filter to be applied, but there will be no label identifying a component of a name.

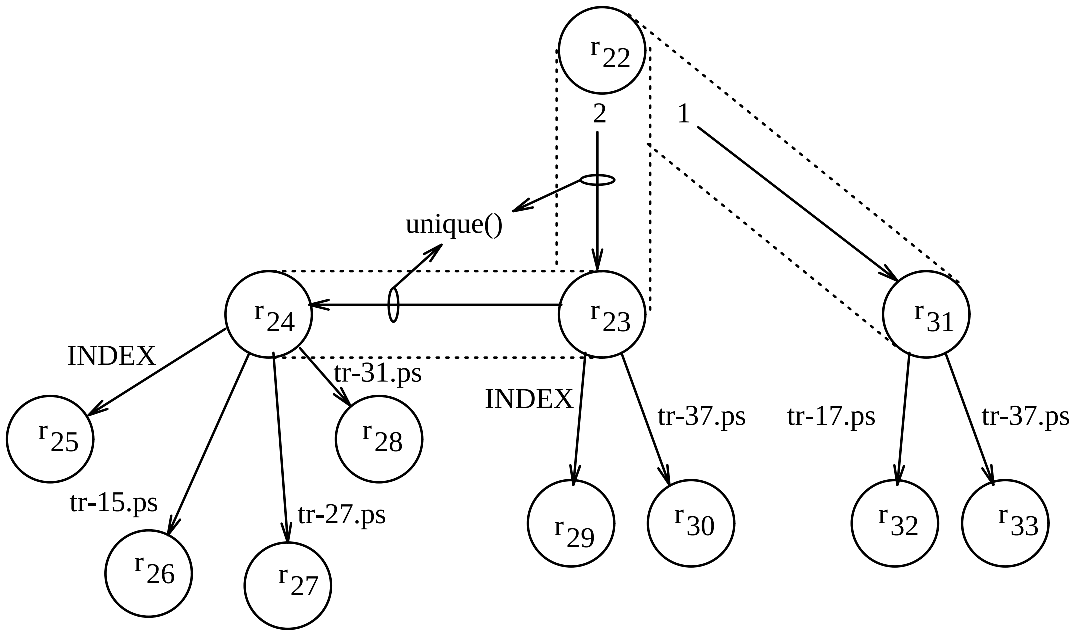


Figure 4.8: The resolution of union links

adds the link to those to be returned, and returns to the top-level transition function. The top-level transition function continues to call itself recursively on the remaining union links, but when it calls itself on r_{23} it applies the **unique** filter, recognizes that it has already found a match, and immediately returns. Had the **unique** filter not been attached to the union link, it would have found a second link named `tr-37.ps`, added it to the list of links to be returned, recursively called itself on directory r_{24}, found no match, then returned through the top-level transition function with two matches.

4.3.2 When is a Naming Network Well-Formed?

Filters and union links are powerful mechanisms for customizing name spaces. Together, however, they can sometimes be too powerful. Because the proper functioning of a system depends on its naming mechanism, it is important that the naming mechanism be well behaved. In particular, it should always return an answer or signal an error, and it should return complete results. A *well-formed* naming network is one in which the resolution of names is guaranteed to terminate with complete results.

Unfortunately, not all naming networks are well-formed. For example, a naming network might include filters that loop indefinitely and it is not possible for the system to decide whether a filter will eventually terminate [Turing 36]. As a practical matter we can require that filters terminate on all inputs, and we can make it the responsibility of the writer of a filter to write it in such a way that it does. In practice, the undecidability of the halting problem does not present a serious problem for the Virtual System Model: implementations can impose the more restrictive constraint that filters must terminate within a specific number of steps or a preset time limit. Such a constraint may result in the rejection of filters that might eventually terminate, but such filters are not likely to be of much use anyway.

Cycles of union links present another potential problem for the correct resolution of names. If a cycle of union links includes a filter that affects the subsequent expansion of a union link, new links might be added each time around the cycle. In such a situation, the complete expansion of the cycle might not terminate. In Subsection 4.3.1 we noted that if while expanding a union link a vertex is reached that has already been visited during resolution of the current component, the context will not be expanded further. This pruning will cause the expansion to terminate, but possibly with an incomplete result. We address this problem by imposing the requirement that cycles composed entirely of union links not include filters. Without intervening filters to change the result of subsequent expansion, the pruning does not affect the completeness of the expansion since the result of a second expansion would be exactly the same as the first. In practice, the transition function can keep track of intervening filters, and if there exists an intervening filter other than the identity filter, an error can be returned. While the method employed to detect cycles is sufficient to guarantee that the recursive expansion of union links terminates, the method does exclude some cycles that might otherwise be valid.

Having discussed what it means for a naming network to be well-formed and having described sufficient constraints on filters and cycles of union links so that a naming network will be well-formed, I now show that the resolution of names will terminate correctly in such a naming network. For the purpose of this analysis, I will assert that the naming network has a finite number of vertices and that each name has a finite number of components.

A name is resolved by successively applying the transition function to the components of a name until no components remain, then applying the access function to the result of the last application of the transition function. To show that the resolution of names terminates, it must be shown that the application of the access function and the transition function will terminate and that, given that these functions terminate, so will the name resolution procedure.

Assertion 4.1 *The name resolution procedure will terminate if the access and transition functions terminate.*

I will show this by induction on the number of components in a name.

Assertion 4.1.1 *Name resolution for a zero-component name will terminate if the access function terminates.*

To resolve a name of zero components, the access function is applied to the real part of the current context and the result is returned, perhaps together with the functional part of the current context[14]. As long as the access function terminates, so will the name resolution function.

Assertion 4.1.2 *If name resolution terminates for names with* n *components, and if the transition function terminates, then name resolution terminates for names with* n + 1 *components.*

To resolve a name of *n + 1* components, the transition function is applied to the first component of the name and the current context. This returns a new context. The

[14]If the current context is a directory containing a union link, the union link is not expanded. The result of name resolution is a reference to the directory, not its value.

remaining n components of the name are then resolved starting from the context returned by the transition function. Thus, the resolution of a name with $n + 1$ components terminates as long as the transition function terminates, and the resolution of names with n components terminates.

By induction on the number of components, the resolution of names with any number of components will terminate as long as the access function and the transition function both terminate on all inputs.

Assertion 4.2 *The access function terminates.*

The access function can be implemented as a lookup in a table with a finite number of entries, each corresponding to one of the vertices in the naming network. Such an implementation will clearly terminate.

Assertion 4.3 *The transition function terminates.*

All that is left to be shown is that the transition function terminates. There are three steps in the application of the transition function. In the first step, the functional part of the current context is applied to the real part. The second step is a table lookup in a table with a finite number of entries[15]. The third step is the recursive application of the transition function to any contexts referenced by union links.

Restriction 4.3.1 *Filters must terminate on all inputs.*

This restriction is one of the constraints we placed on filters earlier in this section. If the constraint holds, the first step is guaranteed to terminate.

Assertion 4.3.2 *If a filter terminates, the table lookup will terminate.*

The table lookup within the transition function will terminate as long as the table has finite length. The length of the table is limited by the number of entries in the real

[15]Each entry corresponds either to an edge in the naming network, or to an entry that resulted from the application of the functional part of the context.

part of the directory, and by any entries that are added by the filter. The number of entries in the real part of the directory is finite, and the number of entries added by the filter must also be finite (if it weren't, the filter would not terminate). Thus, the table lookup will terminate if the filter that is applied also terminates.

Assertion 4.3.3 *The recursive application of the transition function to union links terminates.*

The last step in the application of the transition function is the recursive application of the transition function to contexts referenced by union links. The naming network might include cycles consisting solely of union links. Because the transition function keeps track of the vertices that have been visited while resolving the current component and will not further expand them, the depth of recursion is limited to the number of vertices in the naming network, which is finite.

Restriction 4.3.3.1 *Cycles of union links may not include filters.*

This restriction is the second constraint we placed on the naming network earlier in this section. If the constraint holds, pruning the expansion of union links will not affect the correctness of the result since cycles of union links will not include filters other than the identify filter, and subsequent expansion would yield links that were already obtained during the initial expansion.

To summarize, in a well-formed a naming network the resolution of names is guaranteed to terminate correctly. A naming network in which filters terminate on all inputs and cycles of union links have no filters, is well-formed. It is the responsibility of the writers of filters, and those modifying the naming network, to satisfy these constraints. In practice, implementations may place more restrictive constraints on filters in order to allow the system to enforce compliance.

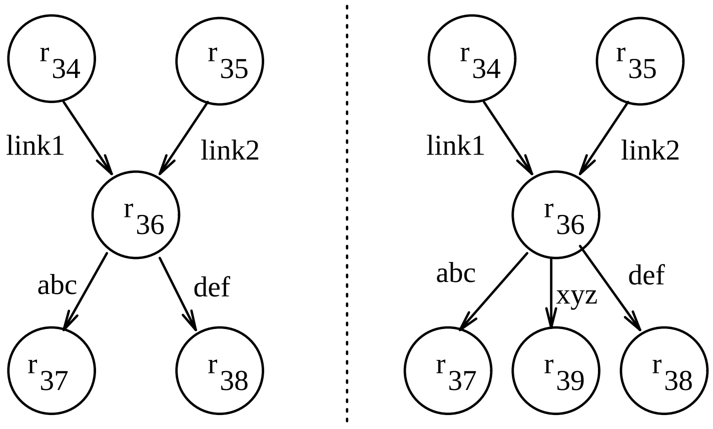


Figure 4.9: Naming network before and after modification

4.3.3 Two Types of Change

Information in computer systems is dynamic. Models for naming in such systems must correctly explain the effects of change. For models that support a single name space, the effects of change are straightforward. Changes are always visible. Once a model supports customization, however, two types of changes must be considered: modifications, which are visible everywhere; and customizations, which are visible only in certain instances.

In the Virtual System Model a *modification* is a change to the real part of a context. Because it affects a vertex in the naming network, a modification is visible in any directories (contexts) that share or include (through union links) that vertex. Figure 4.9 shows a modification to directory r_{36}. The figure on the left shows the naming network before the modification. The figure on the right shows the naming network after the link `xyz` has been added. Note that the change is visible whether the directory is viewed through `link1` or through `link2`.

A *customization* creates a new view of a directory. A customization is visible only when a directory is viewed through a particular link. A customization does not affect the real part of the directory to which it is applied. For example, Figure 4.10 shows the results of customizing the view of directory r_{36} as seen through `link2`. Note that no change was actually made to directory r_{36}. In this case a new directory, r_{40}, was

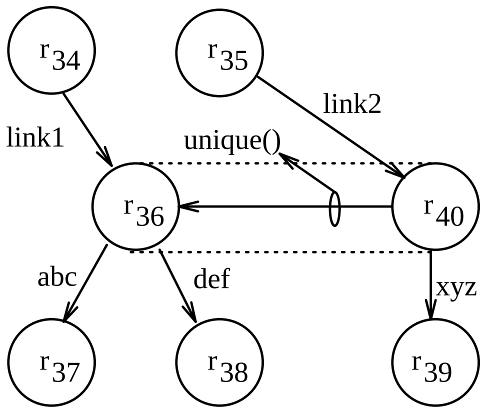


Figure 4.10: Naming network after a customization

created, the links in r_{36} were included in r_{40} through a union link, and directory r_{35} was changed to show the new target of the link named `link2`. A customization might also be performed by adding a filter to `link2`. Because the name of the filter is part of the link, and links are part of directories, such a change would also be a modification to directory r_{35}, not r_{36}.

A customization always results in a modification, but as this example demonstrates, the modification is to a different directory. In this case it was a modification to a directory containing a link to the customized directory. It is possible that the change to the immediate "parent" might itself be a customization, in which case the modification would appear even "higher" in the hierarchy. It is expected that implementations will provide tools to recursively create customizations automatically when requested by the user. The user would specify the change to be made, and the name of the directory which is to receive the modification. By default, the deepest directory writable by the user would be modified and deeper non-writable directories would be customized.

4.3.4 Closure

A formal specification of closure was presented in Chapter 2. Closure is represented the same way in the Virtual System Model. To review, a *closure* function, $\kappa : object \rightarrow NS$, maps an object to the starting context from which names embedded in the object are to be resolved [Saltzer 78, Neuman 89a].

A file system based on the Virtual System Model is different from other file systems in that closure information is required to be stored with each object. In systems supporting a single name space, such information is not necessary since the result of applying the closure function is constant. In other systems, closure information is associated with the running process or active user.

4.4 Relationship to Existing Systems

Features similar to those in the Virtual System Model are supported in several existing systems. In this section I look at each of the features of the Virtual System Model and discuss the relationship to similar features in existing systems.

4.4.1 Directed Graph Based Naming

The structure of the naming network in the Virtual System Model is similar to that of the capability-based directory service used by Amoeba [Tanenbaum et al. 90, van Renesse 89]. The two are similar in that they support multiple name spaces, and that no constraints are placed on the topology of the naming network. Like the Virtual System Model, the active name space in Amoeba is specified by identifying the root directory. A capability in Amoeba corresponds to a link in the Virtual System Model.

Amoeba differs from the Virtual System Model in several respects. Although it supports customized name spaces, Amoeba does not provide mechanisms to specify customized views of directories as functions of other views. Another difference is that the Amoeba directory service supports a user-centered name space. A different initial capability is employed by each user. Thus, closure in Amoeba is based on the identity of the user, and not on the object that includes the name. Finally, capabilities in Amoeba both identify an object, and specify access rights. As such, one's permissions to access an object might be different when the object is named through different paths. The linkage of protection and naming can hinder the sharing of names.

4.4.2 Filters

The function of filters in the Virtual System Model is similar to the domain-switching portal mechanism found in the Universal Directory Service [Lantz et al. 85]. A portal is a call to a separate name server which may have a non-standard implementation, enabling it to resolve names in a manner different than that in a standard name server. An important difference is that a filter is associated with a link. This allows the individual making a link to specify the customization that is to be applied. A portal is written by the creator of the directory, and as such, it is not customizable by the individual users of the directory.

There are also similarities between the functionality of filters, and that of watchdogs [Bershad & Pinkerton 88]. Watchdogs are an extension to the file system that allow the owner of a file to define alternative implementations of file system calls, perhaps returning computed results rather than data that actually appears in the file. Because Unix directories are implemented as files, watchdogs can appear to alter the contents of a directory. Like portals, however, watchdogs are written by the owner of a directory; they are not specified by the individual users. This is an important distinction. With watchdogs, the same watchdog is always applied, while in the Virtual System Model, the filter to be applied to a directory can be different when the directory is accessed through different paths.

Filters can also be applied to files. If a filter is associated with a link to an object other than a directory, the name resolver can optionally return a reference to the filter together with the value returned by the access function when applied to the real part of the final context. Because such a filter would specify alternative access semantics for the file, it would resemble a watchdog more that it would a directory filter. As is the case with directory filters, a file filter is specified on a link. As such, different filters can be applied when the file is accessed through different paths.

4.4.3 Union Links

Union links are related to several mechanisms found in existing systems. Among these mechanisms are search paths and revision control.

Union links can be used to implement search paths. For example, by creating a directory called `/bin` and adding union links to each directory in the search path, the system can search for programs by looking in `/bin`. By associating the **unique** filter with each of the union links, only the first match (in the order the union links were specified) will be returned. Users can add their own programs to /bin directly and they will be found in place of other programs with the same name found later in the search path. In traditional systems where search paths are maintained as part of the state of a running process, problems arise when names are resolved by processes that do not share the same search paths. This problem is not present when search paths are implemented by union links; the search path is part of the name space that is carried along with objects through closure.

Perhaps one of the earliest uses of a mechanism similar to union links was for version control in the Oxford experimental operating system OS12 [Black 91], a successor of OS6 [Stoy & Strachey 72]. A directory in OS12 can contain a link to a single *predecessor* directory, which might itself have a predecessor. If no match is found in the current directory, and if a predecessor directory exists, the predecessor directory is searched. This provides a subset of the functionality of union links. While useful for version control, supporting mechanisms such as search paths requires the ability to specify multiple directories to be searched.

A similar mechanism is used for version control in the 3-D file system [Korn & Krell 90]. In the 3-D file system, the directory seen by the user should be thought of as a stack of physical directories. When resolving names, if no match is found at the current level, the directories below the current level are searched in order of increasing depth until a match is found. Like the predecessor directories in OS12, the stacking of directories in the 3-D file system supports only a subset of the functionality of union links.

The function of the union mount in the Plan 9 operating system [Presotto et al. 91] is close to that of the union link. Plan 9 allows a process to associate multiple file systems (or directories) with a single name prefix[16]. The union mount can be used to support search paths, but like search paths, the union mount is only visible to the process that made the mount and to its children, and it lasts only as long as that process (or its children). Because of its lack of persistence, and because it is not visible to other processes, it is difficult to share parts of a name space that are formed by a union mount.

4.4.4 Closure

The concept of closure comes from the binding of variable names in statically scoped programming languages. Its use for naming in file systems has been discussed [Saltzer 78], but no existing systems (other than Prospero) fully implement it. In many systems, users approximate closure by setting environment variables before running a program. For example, makefiles (scripts used to build large software systems) often specify the names of directories in which libraries and "include" files are to be found.

4.5 Summary

This chapter presented the Virtual System Model. Among the features of the model are a customizable name space, tools for constructing new name spaces, support for synonyms, and a method for selecting the appropriate name space when resolving names. The model was presented formally, and the relationship of the model to features found in existing systems was discussed.

Using the Virtual System Model, the user sees a name space as a hierarchy, but it was shown how the name space is really represented as a generalized directed graph; it only appears hierarchical when viewed from a particular starting node which distinguishes the name space from other name spaces.

[16]Associating a file system with a file name prefix is known as mounting the file system.

This chapter has described the properties of the Virtual System Model by showing how they apply to the naming of files. The Virtual System Model is not just a model for naming files; it is a model for building large systems. As such, the model needs to be applied to all aspects of a system. By using the name space to identify the users, servers, services, and processors that are to be part of a virtual system, the concepts of the Virtual System Model can be extended to these subsystems. The next chapter shows how the Virtual System Model can be applied to many aspects of distributed systems. Among the applications that will be discussed are the organization of information, authentication, authorization, and the selection of processors, services and applications.

Chapter 5

Use of the Virtual System Model

In this chapter I show how the Virtual System Model makes it easier for users to find information and resources in large systems. I start by discussing some of the methods by which information is organized into multiple views. Next, I show how users can construct their own views of the information and resources that are available and how they can make use of the views of others when doing so. It is intended that some views be created specifically for use by others; the incentives to do so are discussed throughout this chapter.

By presenting examples of how the Virtual System Model can be used, I show that it is a powerful and flexible model for organizing large systems. In particular, I show that many mechanisms used to organize off-line information (e.g., papers, books) map nicely to tools that can be implemented easily within the model. I also show how it is easier to keep information up-to-date when these mechanisms are implemented in such a manner.

The Virtual System Model affects more than just the file system. In the second half of this chapter I show how the Virtual System Model affects other aspects of distributed systems including authentication, authorization, and the selection of processors, services and applications.

5.1 Organizing Information

The Virtual System Model allows information to be organized in many ways, and many parties will play a role in doing so. Among the entities that will organize information will be individuals, professional societies, libraries, governments, commercial indexing services, or any collection of individuals sharing a common interest. The subsections that follow describe some of the ways that information can be organized using the Virtual System Model. An important feature of the model is that the same information can be organized in multiple ways, and a reference to an object can appear at more than one place in a name space.

While it might at first appear that some of the organizational methods described in this section could be implemented using symbolic links in traditional systems, it is important to keep in mind that a symbolic link would no longer work if its target moved, that each link would have to be put in place individually, and that in a system as large as the Internet it would be almost impossible to establish consensus on which directories should appear near the root of the hierarchy, and what files should appear in each subdirectory.

5.1.1 Project Organization

It is expected that a project will have an associated virtual system and that a project's virtual system will also be included in the virtual systems of the members of the project. A project virtual system provides a single place where members of the project can look for information related to a project, even if the information is scattered across multiple systems or maintained by different members of the project; it can be included in the virtual systems of new users that join the project, allowing the new users to quickly incorporate the project's view of project-related information into their own view; and it provides a single point of reference from which users that are not part of a project can look for information. When individuals that are not part of the project are to see only a subset of the information related to the project, protection mechanisms can be used to allow non-public information to be hidden. Using a project virtual system in this manner eliminates the need to maintain a separate external view of the project.

The members of a project have several incentives for organizing a project virtual system. First and foremost, they themselves benefit from having a single location in which project-related information can be found. One of the problems that often arises with projects that involve more than one individual is that sources or copies of sources are maintained in directories belonging to the individual members of the project, making it difficult for other members of the project to find them, and possibly resulting in changes being made to old versions. A second incentive applies if a project chooses to make parts of its virtual system available to others, and if links to it appear in places where individuals wanting information about the project are likely to look. Such an arrangement both increases the visibility of the project and reduces the number of queries that must be answered from individuals wanting information about the project.

5.1.2 Index by Author

The individual in the best position to keep track of the papers written by a particular author is that author. It is expected that authors will maintain directories with references to their own works, or at least to those works that they want others to find. The incentive for doing so is visibility. The ease with which others can find one's writings affects the likelihood that those writings will be used. By maintaining one's own index of papers, one can also add cross references to more recent work as it is completed.

The usefulness of such a directory is greatly enhanced when it is itself referenced from a higher-level directory of authors. Such directories are maintained today in library card catalogs and in reader's guides to the literature, but the job of maintaining such directories is greatly simplified when implemented using the Virtual System Model; the maintainer of the higher-level index would only have to update the directory when new authors are added. Once added, it is up to the authors themselves, or to individuals maintaining directories on behalf of the authors, to keep the list of the author's publications current.

5.1.3 Topical Indices

Directories organizing information by topic will help users identify information of interest. Such directories might be maintained by the experts on the topic. Professional societies or libraries might solicit experts in particular areas and create higher-level directories that map topics to the directories maintained by individual experts. For such directories to be useful, their maintainers will have to include references to only those papers that are considered worthy of attention by the community at large, and the process of deciding what is to appear in the directory will constitute a form of peer review. Users who disagree with the choices made by those maintaining such directories are free to create their own directories, or to use similar directories maintained by other individuals. In fact, there will not be an official moderator for each area. Instead, experts will be recognized by the relative numbers of users that employ their views, and by endorsement through inclusion in the higher-level indices maintained by libraries or professional societies.

5.1.4 Indices by Attribute

Directories with references to files with selected attributes can help users find the files they are looking for. Among the attributes that can be used are owner, writer, keywords, file type, and even whether the text of a file contains a particular word or combination of words. User-defined attributes may be stored along with files, and filters that read the attributes or contents of a file can be used to automatically maintain such directories.

Using filters that read the attributes of individual files is inherently inefficient because files that don't match the criteria must also be checked. An alternative approach is available when there exists a precomputed database of object attributes. A filter can map file names to database queries, and the results of a query can be presented as a directory containing those objects that matched the specified attributes.

This latter approach provides a mechanism for making existing databases available though the Virtual System Model (the Virtual System Model does not specify how these databases are to be created or maintained). Many databases with information about object attributes already exist and the Virtual System Model provides a mechanism for tying them together. Some of these databases, library card catalogs for example, contain information about objects that aren't accessible electronically. References to such objects (e.g., books) can be represented on-line as objects whose access method involves physically retrieving the item.

5.1.5 Personal Organization

Users will build their own hierarchies of files by creating directories, subdirectories, and files of their own, and adding links to files, directories and subdirectories created by others. Files that are frequently accessed by a user will probably have short names while names will be longer for objects of less interest. Because directories of other users will be accessible from the user's virtual system, the virtual system will probably contain files that a user has never accessed and might not even know about. These files, however, will be deep in the user's hierarchy.

When users link directories to their own virtual systems, it will be possible to add information to the link to specify how much of the merged hierarchy is to be included. It will also be possible to use filters to customize parts of the attached hierarchy.

Over time, multiple communities of users will evolve. It is expected that the members of each community will have similarly structured name spaces, but name spaces may vary widely across different communities of users. For example, members of the computer science community might organize virtual systems in one way while members of the medical community might think of the world in a completely different manner.

5.2 Looking for Information

Once information has been organized, users can look for it in many ways. A user looking for a paper on heterogeneous computer systems for which Ed Lazowska was an author might find the paper in a directory of papers by Ed Lazowska. A user that didn't know any of the authors might find the same paper in a directory of papers on distributed computing.

Just knowing that the information of interest exists in a published paper can be a big help; many times a user won't even know that. The following subsections show how the Virtual System Model can help users find information when they know little about what they're looking for.

5.2.1 Browsing

The directories and files that a user maintains will be owned by that user. Parts of a user's hierarchy, however, may be owned by other users. Access control information is maintained along with each file or directory, and with each directory link. This information determines who is allowed to read the file or search the directory. It is expected that users will make parts of their hierarchies accessible to others, but how much will be decided by the individual.

One way that information can be found using a file system organized with the Virtual System Model is through browsing. An individual interested in a particular topic can connect to the virtual system of someone else who is known to be interested in that topic, or perhaps to the virtual system for a related project. The user could then look through those virtual systems for documents or files of interest. Of course, he would only see those files that the owner of the virtual system has authorized him to see. The inability to see some information, though, will weed out the information that would not be of interest anyway.

In order for browsing to work best, virtual systems owned by projects should include a directory containing links to related work. Users should also maintain directories containing links to files that they consider interesting or relevant. Files in which others would (or should) have little interest should be kept from view by applying appropriate protections.

Browsing is considerably more likely to be effective under the Virtual System Model than in traditional file systems. The Virtual System Model encourages users to make their own links to the files in which they have an interest. As such, interesting files are likely to appear in the hierarchies of multiple people, thus increasing the likelihood that they will be found by browsing.

5.2.2 Finding Things

In today's society, if something is available that is of interest, it is usually found through directories such as the phone book or yellow pages, through reading newspapers and other periodicals, or by word of mouth. In the computer science community, these sources of information are supplemented by technical papers, electronic mail, and mailing lists. These methods of discovery are natural, and it is likely that they will continue to find significant use even once other mechanisms are in place.

The Virtual System Model allows much of the information that is useful for finding objects, but which to date could only be obtained by external means (such as asking the author of a paper), to be included as part of the file system. A virtual file system provides a matrix through which users can navigate to find the desired information.

5.3 Relationship to Distributed Systems

So far this chapter has concentrated on the ways that the Virtual System Model can be used to organize information in large systems. The Virtual System Model affects all aspects of a distributed system. The model can be applied to other aspects because naming is important to all of them. The system components that are to be part of a virtual system can be specified by creating special links in the name space that reference those objects. By providing a mechanism to identify the components that are of interest, it is possible to tailor the behavior of the system. This section shows how the model can be applied to authentication and authorization, processes and processors, and services and applications.

5.3.1 Authentication & Authorization

Authentication is used to ensure that an entity claiming to have a particular name is, in fact, the entity to which that name refers. Authorization is used to make sure that a named entity is allowed to perform a particular operation. For both to work, there must be an accepted mapping from a set of names to entities requiring authentication. Although this mapping does not need to be the same in all places, authorization is significantly simplified if it is. Since virtual systems overlap, allowing the use of non-unique user-level names for authentication could create parts of the system where two different sets of names are in use. This would complicate access control for objects in those overlapping regions. For this reason, authentication and authorization will be based on the globally unique names of users. This does not preclude having an additional level of indirection through which users specify names.

It is expected that authorization will be accomplished primarily through access control lists (ACLs). Support for groups allows ACLs to provide a flexible interface for access control. Groups can contain users or other groups. As with names, groups should be represented uniquely within ACLs. Users, though, might use different names to identify them. A problem arises in deciding how to display the content of ACLs to the user. With groups, it is perhaps even more important for the user to see a name with which he is familiar. Mapping back from the unique name to a local name might be costly, however.

One solution is to give the user the option of how ACLs are to be displayed. The typical user would choose the local names. Since typical users are likely to have only a small number of users or groups defined within their virtual system, the mapping might not take as long as for an advanced user. In any case, information is needed to help users figure out what the purpose of a group is if the group does not map to a local name. Each group can have a comment associated with it. These comments can be displayed next to the local or unique name when the ACL is listed.

So far, I have discussed the naming side of authentication, and how these names are used for access control. Authentication itself has not been talked about. One of the problems that arises as a system spans multiple organizations is lack of trust. Services can't be required to trust every authentication server that exists. It might not even be the case that there is a single authentication server trusted by everyone. A particularly paranoid application might only trust authentication performed independently by its local authentication server.

Another issue is the distribution of authority. Unless, for every communicating pair of entities, there exists some trusted entity sharing a secret with both parties, then multiple authentication servers might be involved in authenticating a principal.[1] It is important that the end service know which authentication servers were involved when deciding whether to trust the authentication [Birrell et al. 86].

There may be different competing collections of authentication servers. Such servers might differ in who they are run by, or perhaps even in the protocol required for initial authentication (how the user logs on). It should be possible, from within each virtual system, to decide which authentication servers are to be trusted. When accessing a particular object, the authentication servers to be trusted would be those specified in the virtual system linked to the object though closure.

5.3.2 Processes and Processors

A virtual system may include references to a collection of processors. If specified, these will be the processors on which applications will run. Processors within a single virtual system may be of different types, and the processor chosen will be based on the requirements of the application as well as on other information available at the time (such as load, etc.). Applications will be able to choose processors from within the virtual system on which they are running, within which they were installed, or alternatively, on other processors that they find dynamically.

[1]Even with public key systems, the public key to party A must be known to a key repository whose public key is known by B.

When a process is created it is associated with a virtual system. Typically the associated virtual system is that defined by the name space to which the running program has been bound by closure. If the virtual system specifies the processors (or other resources) to be used by the application, then those resources are used, either in place of or in addition to the resources available to the user. The optional use of closure for the selection of the processor or processors on which an application will run allows applications that require a special processor or processor configuration to be run automatically on an appropriate machine, even if the user is unaware of the special requirements.

Just as the associated virtual system can be used to restrict the processors on which an application can run, it can also be used to restrict the objects that can be accessed by the process. This can be accomplished by setting a flag indicating that processes running within a particular virtual system should only be able to access objects that are part of that virtual system, and that the ability to switch to other virtual systems should be disabled. By building a virtual system with limited size, and by protecting the virtual system so that it can't be modified from within, processes can be run in a protected environment. In order for this to work, the processor on which the process executes must enforce the restrictions imposed by the virtual system. By suitably restricting what a process can read and write, the Virtual System Model can play a role in the implementation of mandatory access control policies.

5.3.3 Services and Applications

In existing systems, services and applications differ in the way they are accessed. A service typically provides basic operations and is accessed across a network. An application is typically a local program, but it may provide a front end for one or more services. In the Virtual System Model, applications can be associated with a set of processors on which they may run. When a user runs an application, it might run remotely. This will blur the distinction between an application and a service. The distinction will become more a matter of whether the interface is intended for humans, or for other programs.

Given the name of a service or application, clients will use the directory service to find the server or executable. If multiple instances exist, the user will have to make a choice. This choice will be specified by including the selected server or set of servers in the user's virtual system. The use of filters provides a useful tool for choosing a server. If a link is made to a directory containing pointers to instances of a service, a filter associated with the link can be used to filter out all instances that do not satisfy the client's constraints (e.g., price).

5.3.4 Tying it All Together

Figure 5.1 shows a complete virtual system. The resources available over the network include three workstations, two multiprocessor systems, two authentication servers, and a print server. The dotted line encloses the resources that form one virtual system. Other virtual systems include overlapping resources, but they are not shown here. The virtual system that is shown in the figure includes a subset of the nodes on multi-processor 1, several files from workstations 1 and 2, the processor and display from workstation 2, and the print server.

The virtual system shown is defined by a name space which assigns names to each of the components of the virtual system. The name space is maintained by a distributed directory service, though the directory service is not explicitly shown in this picture since it permeates all parts of the system. It is the naming mechanism, as implemented by the directory service, that has been the principal focus of this dissertation.

Closure is represented by a dashed arrow. One of the files, a program on workstation 3, is associated by closure to the virtual system that is shown. If the program is configured to select processors from the associated virtual system, then that program will run using resources of the virtual system shown. In this example, the program that is associated with the virtual system is not itself part of the virtual system. Thus, when a user of a different virtual system, possibly running on workstation 3, executes the program, it is automatically run remotely, and names are resolved using the virtual system bound to the program.

Finally, protection of resources associated by closure with this virtual system relies on the security services of authentication server 2. Again, it is the files associated by closure to the virtual system that inherit the security attributes of the virtual system, not necessarily the files that are part of the virtual system.

5.4 Summary

This chapter has shown how the Virtual System Model supports the organization of information and resources in large systems. Many of the techniques used to find and organize off-line resources are easily applied to on-line information and services using the Virtual System Model. Using the Virtual System Model, the task of keeping such reference materials up-to-date is made simpler by eliminating duplicated effort and by assigning the responsibility for maintaining information to the parties that are in the best position to do so.

The Virtual System Model affects more than just the file system. Because most aspects of distributed systems depend on naming, the Virtual System Model allows the customization of almost all parts of a distributed system. Among the affected components are authentication, authorization, and the selection of processors, services and applications.

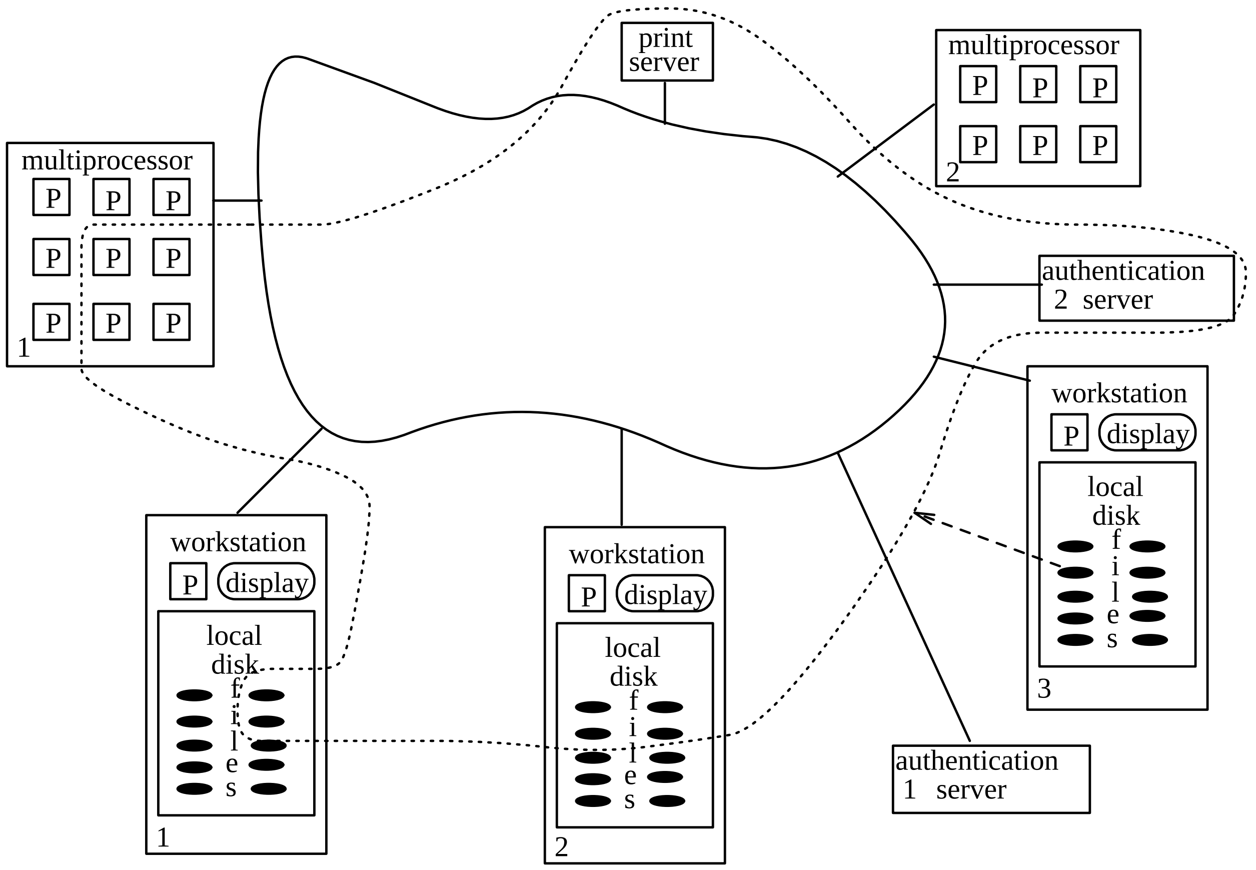


Figure 5.1: A complete virtual system

Chapter 6

A Prototype Implementation

In this chapter I describe Prospero[1], a prototype implementation that applies the concepts of the Virtual System Model to the design of a global file system. A prototype was needed to validate the solutions proposed by the model. A global file system seemed an appropriate test because of the large number of files that already exist, the ability to access files over a large geographic area, and the distribution of files across not just system boundaries, but also administrative and national boundaries.

The chapter begins by describing the function of the prototype. The chapter continues with an overview of the implementation, describing the principal components of the prototype and showing how they fit together. The implementation of filters, union links, and closure is described. A file system has many functions that are not specific to the Virtual System Model, yet these functions must be provided if a file system is to receive widespread use. These functions are discussed and I describe the support provided for them by Prospero. For functions not supported by the prototype, possible approaches are discussed and the steps taken to support future implementation of these approaches

[1]From the *Tempest* by William Shakespeare. Prospero was the rightful Duke of Milan who escaped to a desert island. When his enemies were shipwrecked on the island, Prospero used his power of illusion to separate the party into groups, each of which thought they were the only survivors. Thus, he caused each group to see a different view of the world. As time went on, the shipwrecked parties slowly learned about the others, and thus, pieces of other views were added to their own.

are described. Finally, users won't accept a system that is slow. I conclude the chapter by presenting performance figures for the prototype implementation. Experience with the prototype is described in Chapter 7.

6.1 Function of the Prototype

The Prospero prototype solves a real and recognized problem, that of organizing files that are scattered across the Internet, a network connecting educational, government, and commercial computers worldwide. The Prospero file system allows files that are logically related to be grouped together, even if scattered across multiple systems, and it allows files to be organized in multiple ways.

The Prospero file system differs from traditional distributed file systems in several respects. In traditional file systems, the mapping of names to files is the same for all users. In the Prospero file system, users construct their own customized views of the files that are accessible. A virtual system defines this view and controls the mapping from names to files. Prospero provides tools to support this customization; among these tools are the filter and the union link.

Unlike most file systems, Prospero is concerned primarily with the naming and organization of files, not with the method used to store data or access a file once it has been found. The files available through Prospero are stored in existing file systems and multiple access methods are supported.

Using Prospero, users can organize files in many ways, creating their own directory structures from scratch, or deriving them from directory structures that already exist. When looking for information of interest, users can use the many directory structures that already exist. These directory structures organize files available from Internet archive sites worldwide.

6.2 The Implementation

There are two parts to the Prospero implementation, the client and the server. When names are resolved, the application resolving the name (the client) contacts one or more directory servers on the host or hosts storing the directories that correspond to the successive components of the name. The server looks up the requested name in the specified directory, returning the result to the client. The result is either used to access the named object, or it is further resolved by contacting additional directory servers. The subsections that follow describe the server and client in greater detail; Figure 6.1 (at the end of Section 6.2.1) shows how the pieces fit together.

6.2.1 The Server

The Prospero directory server accepts a directory identifier and optionally the name of the link to be returned to the client. The server returns the links in the directory that match the specified name, or all links if the name was not specified. A Prospero link specifies the name of the host that stores the object and an identifier for the object on that host. These fields can be thought of as an address for the object. The identifier of an object is the information needed by the Prospero server to identify and locate the object on the local system. It is usually a native file name for the object, but it could also be an integer used as an index into a table, or even the arguments to a procedure used for retrieving the object.

A Prospero link specifies other information including whether the link is a union link, and what (if any) filters are associated with it. If the target of the link is a directory, the link provides the information needed to resolve a name in that directory by contacting the directory server on the host specified by the link.

In addition to answering directory queries, a Prospero server answers requests for information about the objects it manages, including system and user defined attributes associated with objects. The values of these attributes can either be strings, used for the LAST-MODIFIED, SIZE, or similar attributes, or links used to represent named cross-references to other objects. Finally, if configured to do so, the directory server will accept changes to directories and to the attributes of the objects it maintains.

The client and directory server communicate using the Prospero protocol, described in Appendix D. To reduce the overhead that would otherwise be incurred when establishing connections to multiple directory servers, the Prospero protocol is layered on top of a reliable datagram protocol described in Appendix E and implemented using the connectionless User Datagram Protocol [Postel 80].

The directory server accepts queries, retrieves the requested data, and responds to the client. Internally, the server uses the Prospero library to process directory information and object attributes. Functions that are specific to the server are supported by the *psrv* library including all reading from and writing to the native file system. In this role, the *psrv* library translates the native directory format into the Prospero directory format, thus making available through Prospero existing files and directories not created through Prospero. The *psrv* library can also translate Prospero file system queries to database queries, allowing a database to be substituted for the native file system's directory structure in support of alternative search methods such as keyword or attribute queries.

The basic function (querying and modifying directories) of the Prospero directory server is similar to that of capability-based directory servers like the one in Amoeba [Mullender & Tanenbaum 86]: the directory server has no idea how its directories fit into the name space; each directory is a separate object that may be referenced by many other directories; and cycles are even allowed. The Prospero directory server differs from those in capability-based systems in that its object handles (links) do not grant authorization to access the object. The Prospero directory server further differs from the Amoeba directory server in that links in Prospero contain information about the storage site for the object, whereas in Amoeba they provide only a unique identifier for the object which must be located using a broadcast mechanism.

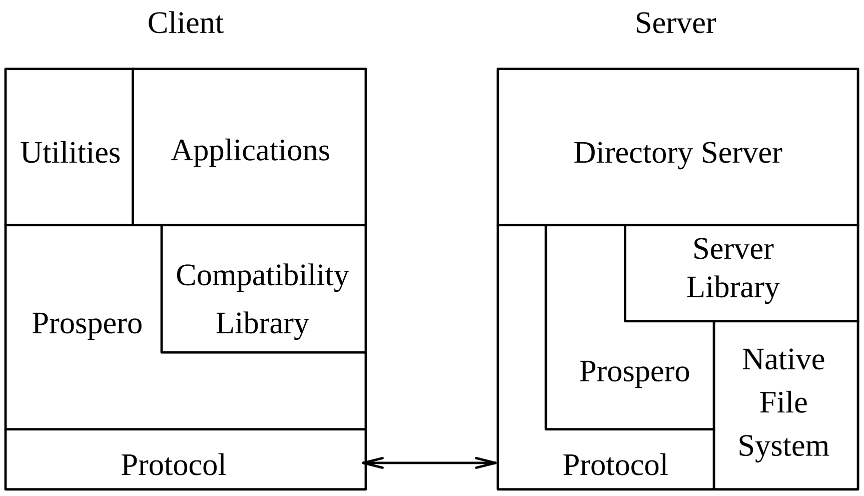


Figure 6.1: Basic structure of the Prospero implementation

6.2.2 The Client

The Prospero client is implemented as a library that resolves names, retrieves attributes, and accepts updates to directory and attribute values. A second library, the *compatibility library*, remaps **open** and several other system calls, allowing existing applications linked with the library to resolve names using the active virtual system. No changes to the source code of the programs are required. The procedures in both libraries are described in Appendix B.

Users access Prospero through existing applications that have been linked with the compatibility library or through utilities that allow them to specify the active virtual system, navigate through the specified name space, and to manipulate directories and object attributes. Instructions for using these utilities, the syntax for specifying file names, and other information of importance to the user are described in the Prospero user's manual which appears in Appendix A.

Resolving Names

The Prospero library keeps references to the current working directory and the root of the active virtual system. When a user or application wishes to resolve a name, the first component is resolved relative to the appropriate directory by sending a query to the corresponding directory server using the Prospero protocol (described in appendices D and E). Successive components of the name are resolved by sending queries to the directory servers named in the links returned by the previous queries. This process is repeated until all components of the name have been resolved[2].

Filters

If a filter is associated with a link, the filter is applied to the result of the directory query before the current component of the name is looked up[3]. A filter can remove links from or add links to a directory, change the names of links, or even change the way a directory hierarchy appears to be organized (e.g., creating subdirectories).

Filters are written in C and are dynamically linked with the name resolver when they are applied. If an appropriate object file does not exist at the time the filter is applied, it can be automatically compiled before loading. The code for the linker is derived from that used by the Extension Interpreter to support extensible user interfaces [Notkin & Griswold 88]. For the prototype implementation, the dynamic linker only runs on the VAX, though it would be straightforward to use a multi-platform linker for improved portability [Ho & Olsson 90].

Although users can write their own filters, most users can get by using predefined ones. Among these are: **flatten** (take a directory hierarchy and make it appear like a single level name space), **match** (pass links matching a specified list of names), **matchhost** (pass links whose target is stored on the matched hosts), **distribute** (create subdirectories for each value of the specified attribute and distribute files among those directories according to that attribute), and **attribute** (pass only links for objects with attributes matching those specified).

[2]An optimization allows a directory server to resolve more than one component of the name at a time as long as all intervening directories are stored on the same server.

[3]This means that the directory server must be instructed to return all links in the directory, not just those matching the component.

Figure 6.2 shows code implementing the **distribute** filter. Declarations and some of the bookkeeping code have been removed to make the example more readable, but the majority of the code is shown. The **distribute** filter is interesting in that it attaches new filters to the links it returns. In fact, the subdirectories "created" by the **distribute** filter are all filtered links to the original directory. The filters are attached to each link with arguments corresponding to the values of the attributes returned by the call to **pget_at**. Duplicates are eliminated during the call to **vl_insert**.

Union Links

At any point in the resolution of a name, the directory server might return one or more union links. Such a response indicates that the directory has not been completely searched, and that the current component of the name should be resolved in the directories named in the union links[4]. If the expansion of union links returns a link whose name has already been returned, then the first link takes precedence and subsequent links are added to a list of conflicts. Since the list of conflicts is ignored in most cases, this behavior implements the **unique** filter. In the prototype, the **unique** filter is applied in this manner to all union links.

Closure

Prospero supports closure by storing a link to the closed name space as the CLOSURE attribute of each object. When a file is opened, its CLOSURE attribute can be read and this value used as the starting point when resolving names read from the file. Some objects, electronic mail messages for example, are passed from system to system by mechanisms external to Prospero. For these types of objects, the closure information must be included as part of the object itself, usually in a header. This works well for electronic mail since the user interface can read the header and use the closure

[4]If the directory is being listed, then the results of querying the union linked directories are merged with the rest of response from the directory that returned the union links.

```
VDIR filter(dir,ip,argc,argv)
    {
        vdir_init(nd);
        sd = vlcopy(dirlink,0);
        avf = rd_vlink("/lib/filters/avalue.o");

        /* Step through attribute values creating subdirs */
        cl = dir->links;
        while(cl) {
            attributes = pget_at(cl,argv[0]);
            for(ca = attributes;ca;ca = ca->next) {
                /* If not first link, then make copy */
                if(nd->links) sd = vlcopy(nd->links,0);

                /* Set name of new subdir and insert it */
                sd->name = stcopyr(ca->value,sd->name);
                if(vl_insert(sd,nd) == PSUCCESS) {
                    /* If successful, then set filter arguments */
                    /* Find last filter on current subdir       */
                    for(avf=sd->filters;avf->next;avf=avf->next);
                    sprintf(farg,"%s %s",ca->aname,ca->value);
                    avf->args = stcopyr(farg,avf->args);
                }
            }
            atlfree(attributes);
            cl = cl->next;
        }

        /* Return the result in the original directory*/
        vdir_copy(nd,dir);
        return(dir);
    }
```

Figure 6.2: Simplified code for the distribute filter

information in an appropriate manner, while stripping off such headers before the message is displayed by the user.

Accessing the Named Object

The mechanisms described so far are used to locate a named object. If the object is a file, a method is needed to retrieve its contents. Because Prospero runs in a heterogeneous environment, it was necessary to support multiple access methods. Among those presently supported are Sun's Network File System [Sandberg et al. 85], the Andrew File System [Howard et al. 88], and access using the anonymous file transfer protocol [Postel & Reynolds 85]. When the client has located a file to be opened, an additional query is sent to the directory server on the system storing the file. This query requests the value of the ACCESS-METHOD attribute. The directory server returns a list of the methods it supports (in order of preference). The client chooses one of the methods and opens the file[5].

References to objects on hosts that do not run Prospero can be included in a virtual file system through *external* links. For external links, the access method is encoded as part of the object type field of the link. When a file referenced through an external link is opened, the access method encoded in the link is used and no negotiation of access method occurs.

6.3 Other issues

This section describes various operational issues that are raised by the Prospero file system. Some of these issues are addressed by the prototype while others will be addressed at a later date. Where an issue is not addressed by the prototype, possible solutions are proposed, and the steps taken in the prototype to enable these solutions are described. The operational issues that are discussed are protection and security, mobility, garbage collection, replication, and autonomy.

[5]If the anonymous FTP method is chosen, the file is first retrieved, then the local copy is opened. The anonymous FTP method is supported for read access only.

6.3.1 Protection and Security

Protection of objects named by Prospero is based on the protection mechanisms employed by the underlying access method. The ability to resolve the name of an object does not grant permission to access the object. Protection of directory information, however, is based on access control lists that are associated with directories and with individual links within directories. The authorization mechanism provided by Prospero applies to the ability to resolve names in or to modify a directory. It does not apply to the referenced object itself[6].

Access control lists in Prospero assign rights to principals (users, groups, programs, etc.) identified by various authentication mechanisms. The authentication method presently supported identifies the host from which a request originates and the name of the user on the host. It trusts the software at the client to correctly identify the user. Hooks are in place to support stronger authentication methods such as Kerberos [Steiner et al. 88].

Table 6.1 shows the rights that appear in access control lists. The entries on the left appear in directory ACLs and apply to all links within the directory. The entries on the right apply to individual links within a directory. If an entry from the right appears in a directory ACL, then it applies to all links in the directory that do not specify individual ACLs. Negative rights may be assigned by preceding the permission list with a "-". Such an entry will deny rights to a principal that would otherwise be granted rights by a subsequent ACL entry. The add-rights and remove-rights permissions are a restricted form of the administer permission that allow the authorized principal to grant or remove only those rights that follow the add or remove symbol in the permission list.

[6]Though the attributes of an individual object might include the access control information used by the underlying access method.

Table 6.1: Protection modes in access control lists

Directory		Link	
Character	Meaning	Character	Meaning
A	Administer	a	Administer
V	View	v	View
L	List	l	List
R	Read	r	Read
M	Modify	m	Modify
D	Delete	d	Delete
I	Insert	does not apply to link	
B	Administer	does not override link	
Y	View	does not override link	
>	Add-rights	]	Add-rights
<	Remove-rights	[	Remove-rights
)	Add-rights	does not override link	
(	Remove-rights	does not override link	

6.3.2 Garbage Collection

In the Virtual System Model, all links to an object are equivalent, and an object continues to exist as long as there exists a link that references it. In practice, this is difficult to support in distributed systems. The obvious approach is to maintain a count of the links that reference each object, and the storage for the object can be reclaimed when the reference count reaches zero. Unfortunately, this approach has two problems. First, in a loosely coupled distributed file system such as Prospero, a server failure might result in the loss of a link to an object without a corresponding decrement of its reference count. Second, the topology of the Prospero naming network allows cycles, and the reference counts for objects that are part of a cycle will be non-zero whether or not the cycle is reachable from the rest of the graph.

The primary method used for garbage collection in Prospero is timeouts. Every object has an associated time-to-live (TTL). The TTL is the period for which the object is guaranteed to exist. When a link is made to an object, the TTL is added to the current time, and the resulting link expiration date is stored with the link. To guarantee that a link will continue to work, it must be refreshed before its expiration. This can be done when the link is referenced, but for links that are infrequently referenced, a background process can refresh the link before it expires.

If a user wishes to recover the storage space for a file, the file is flagged for removal. When links to the object are refreshed, notification is sent that the object is about to disappear, and anyone wanting to maintain their link must copy the object elsewhere[7]. Subsequent users have the option of making their own copy, or updating their link to refer to one of the new copies.

The TTL for an object is chosen by the owner of the object. The cost of the background refreshing of links will depend on the value of the TTL. The longer the TTL, the fewer refreshes are required. But a user is required to keep a copy of an object until the TTL expires even after the user's link to the object is deleted. This limits the user's ability to reclaim the storage used by the object.

The owner of an object can grant a *backlink* to a directory that includes a link to the object. A backlink is a reference to the link referencing the object. If a backlink has been granted, the grantor guarantees that, before the link is invalidated, the grantor will notify the server that maintains the directory that was granted the backlink. A link for which a backlink has been issued does not need to be refreshed[8]. The drawback is that a very large number of backlinks would be required for a widely referenced object, and the owner of the object can not get rid of it if the holders of any of the backlinks are inaccessible. As such, backlinks are most appropriate for infrequently accessed stable objects with few links.

The mechanism described so far can make sure that storage is not reclaimed for objects that have non-expired references. If a cycle of unreachable objects exists, however, the links will be refreshed periodically by the background process, and the storage will

[7]The original copy must continue to exist until its expiration time since other users with links to the object might not trust the first user to make an accurate copy.

[8]Actually, a TTL does apply to such a link, but the TTL will be much longer than for links that were not granted backlinks.

not be reclaimed. Because there is no global root, it is not possible to require the background process to refresh only those links reachable from a common starting node. The scalability of such an approach also presents problems since every object in the global system would be accessed on each sweep. An alternative is to store an additional timestamp with each object indicating the time it was last referencable. When an object is referenced, the timestamp is set to the time of the reference. When a link is refreshed, it propagates its last referencable timestamp to the referenced object (which replaces it's own if the new timestamp is more recent). The last referencable timestamp for objects in an inaccessible cycle will eventually stabilize at a value that is less than or equal to the time at which the cycle became inaccessible.

The fields to support these garbage collection methods are supported by the Prospero implementation, but the code to automatically refresh the link expiration dates and to propagate the last referencable timestamp has not been implemented in the prototype. Implementing and evaluating the scalability of this approach, and improving upon it, are topics for future research.

6.3.3 Object Mobility

A Prospero link continues to work even if the object it references has moved. Prospero supports object mobility through the use of forwarding pointers [Fowler 85]. A user attempting to access an object at its old location is given the new location of the object and the Prospero library automatically retries the request using the new information. When a link is refreshed, any forwarding pointers are followed and the link is updated to reflect the new address for the object. This reduces the number of steps required for future reference, and it also makes it possible to get rid of forwarding pointers once all possible links with the old address have expired.

6.3.4 Autonomy

The physical systems tied together by Prospero may be autonomous. Each can operate independently from the others. Prospero has no central database, and the naming network need not have critical nodes, though depending on the organization of individual name spaces, some nodes may be more important than others. As expected, one would not be able to access objects or services which reside on systems that are unreachable due to network or hardware problems. To resolve a name, the directory corresponding to each component of the name must be accessible. If an intermediate directory is not available, the object can not be accessed using that name, though it might still be possible to access the object using an alternate name. Autonomy is improved when the directories near the center of a user's name space are stored, replicated, or cached on local systems.

6.3.5 Replication

Large distributed systems should support more than one method of replication. This is especially important if the system is to receive widespread use. It is not even enough to provide a fixed set of replication techniques. Instead, the system should provide the infrastructure necessary so that applications can provide their own replication when appropriate [Neuman 90].

Although Prospero does not presently implement replication, support for replication was an important consideration in its design. Prospero provides the necessary support for replication by allowing a list of replicas, a specification of the type of replication to be used, and any data specifically required by that mechanism (e.g., votes) to be stored in the attribute list for each object.

While the authoritative copy of the replication information is stored with each replica of the object, a possibly out-of-date or incomplete list of replicas can be stored with each link[9], allowing the user to select an alternate if the replica referenced by the link is

[9]A directory can associate more than one link with a single name, and each link is a reference to one of the replicas.

unavailable. The list is treated as a hint, the authoritative information is checked when the replica is accessed. Because it is unlikely that one will know where all the links to an object originate, it is not practical to require that the links be updated when the replication information changes.

6.4 Performance

While performance was not the most important issue in the design of Prospero, it was important that the performance be comparable to that of existing distributed file systems. Users won't use a system that is slow, no matter how much better it is for organizing information.

Table 6.2 shows the performance of the Prospero client on a DECstation 5000. The remote Prospero server is running on a second DECstation 5000 on the same Ethernet. The numbered columns represent the time required to resolve a name with the specified number of components. The second to last column is the time required to negotiate the access method and the final column is the time it takes to open the file. Since Prospero uses the existing access methods of the underlying system, the last column is also the time it takes to open the file without Prospero.

Table 6.2: Approximate time to resolve a name and open a file

	Time to resolve a name in Prospero					Negotiate	Open	
	Number of components					Access	Access	
Storage site	1	2	3	4	5	Method	Method	Time
Remote	38ms	76ms	115ms	153ms	191ms	32ms	NFS	125ms
Local	21ms	43ms	63ms	86ms	107ms	-	Local	27ms

In compiling these figures, the optimization that resolves multiple components at the same time has been disabled. Thus, the time to resolve a name with consecutive components stored on the same server would be less.

The performance of the prototype is more than adequate. The time required to open a file using Prospero (name resolution + negotiation + open) when all components of the file's name are stored on a single server is less than twice that required to open the file directly with NFS. This degradation is to be expected when one considers that at least one extra pair of network messages is involved. For most applications, the frequency of opens is low enough that the increased latency won't be noticed by the user. Because Prospero uses the underlying access methods of the system on which it runs, the time for reads and writes is unchanged.

6.5 Summary

This chapter described the prototype implementation of the Virtual System Model. The implementation has two parts: a server which responds to directory queries, requests for file attributes and accepts updates; and a client which resolves names by contacting the appropriate servers, expands union links, applies filters, negotiates access methods, and opens files.

To be useful, a distributed file system must address many issues other than the storage of files and the resolution of names. A few of the issues of importance are security, object mobility, garbage collection, and replication. The first two issues are addressed by the prototype and the approach used was described. Possible approaches to addressing the remaining issues were discussed, but the implementation, evaluation, and improvement of the proposed approaches is a subject for future research.

Chapter 7

Use of the Prototype

Evaluation of the Virtual System Model is a difficult task. Because the prototype allows users to do new things, comparison to existing systems must be qualitative; there are no numbers to compare. Chapter 6 presented some performance figures, but evaluating the model based on those figures would completely miss the point: while the performance is quite good for a prototype, the real contribution of model is that it enables users to better organize information.

An alternative approach to evaluation would be to point to the prototype implementation and claim that it successfully satisfies the requirements set for the Virtual System Model. While such an assessment would demonstrate that the model can be practically implemented, it too would miss the important issue: whether the model is useful.

To answer this question, one must look at the use of the prototype. There are several questions to be answered: Do people use the prototype? What is it used for? Does the prototype help them organize information, and once organized, does it help them find what they're looking for?

Table 7.1: Systems using Prospero by country

Country	Domain	Count	Country	Domain	Count
Austria	AT	42	Iceland	IS	2
Australia	AU	144	Italy	IT	20
Belgium	BE	3	Japan	JP	106
Brazil	BR	2	South Korea	KR	8
Canada	CA	559	Mexico	MX	8
Switzerland	CH	146	The Netherlands	NL	121
Germany	DE	338	Norway	NO	130
Denmark	DK	21	New Zealand	NZ	17
Spain	ES	8	Portugal	PT	3
Finland	FI	29	South Africa	ZA	1
France	FR	126	Sweden	SE	111
Greece	GR	6	Singapore	SG	42
Ireland	IE	1	United Kingdom	UK	93
Israel	IL	36	United States	See 7.2	5392
India	IN	2	**Total**	29	7518

7.1 Usage Statistics

The prototype, Prospero, has been available since December 1990. As of the end of November 1991, the system had been used from more than 7,500 systems in 29 countries on six continents to find files available from Internet archive sites. The prototype has made it possible to organize information scattered across the Internet into a coherent whole. Table 7.1 shows the number of systems on which Prospero is used in each country. Table 7.2 breaks down the U.S. totals by type of organization.

On a typical day, Prospero is used by more than 1,400 users[1] on more than 1,350 systems to make more than 6,000 queries. Over the course of a week Prospero is used by more than 4,500 users. These statistics are based on the logfiles of only seven Prospero servers. While the user and system counts accurately reflect the use of Prospero as a

[1]I treat requests from users with the same username, but on different systems, as coming from a single user. While this results in an underestimate of the number of users, to do otherwise would result in an even greater overestimate.

Table 7.2: Breakdown of U.S. systems by organization type

Organization type	Domain	Count
Educational	EDU	4160
Commercial	COM	745
Military	MIL	88
Other Government	GOV	287
Network Administration	NET	40
Other Organizations	ORG	72
Total		5392

whole, the query counts are understated since they do not include queries to servers for which I did not have access to logfiles.

It is interesting to note the strong foreign interest in Prospero. More than 25 percent of the sites running Prospero are located outside the United States. In fact, the foreign users more frequently use Prospero's ability to create customized views of the information that is available. I attribute this to the fact that much of the information on the Internet is organized under the assumption that the user will have a high-bandwidth connection to U.S. sites; customization allows foreign users (who often have slower links to the U.S.) to substitute views that are more appropriate for their locale.

7.2 What it is Used For

Prospero has been used to organize information available from Internet archive sites, and to look for information that has been organized or indexed by others. At present, users do not use Prospero in place of their existing file systems; they turn to Prospero when looking for information that they think is available over the Internet.

Users interact with Prospero using commands that move through, modify, and display directories. Users can also interact with Prospero through existing applications that have been relinked with the Prospero compatibility library, browsers and other tools that have been specifically written to use Prospero, and gateways that allow Prospero to be used from other services. Many of the applications that rely on Prospero were written by others.

7.2.1 Using Prospero Itself

As distributed, a user's virtual system starts out with links to directories organizing information of various kinds in several ways. Figure 7.1 shows a sample session with Prospero. Users find information by moving from directory to directory in much the same manner as they would in a traditional file system. Users do not need to know where the information is physically stored. In fact, the files and directories shown in the example are scattered across the Internet. At any point, a user can access files in a virtual system as if they were stored on his or her local system.

In the example, the user connects to the root directory and lists it using the `ls` command. The result shows the categories of information included in the virtual system. The information includes online copies of papers (in the papers directory), archives of Internet and Usenet mailing lists (in the mailing-list and newsgroups directories[2]), releases of software packages (in the releases directory), and the contents of prominent Internet archive sites (in the sites directory). Files of interest can appear under more than one directory. For example, a paper that is available from a prominent archive site might also be listed under the papers directory.

Next, the user connects to the papers directory, lists it, and finds the available papers further categorized as conference papers, journal papers, or technical reports. The technical report directory is broken down by organization, and by department within the organization. The journals directory is organized by the journal in which a paper appears, and the two journals that are shown are further organized by issue. Use of the `vls` command shows where a file or directory is physically stored, demonstrating the fact that the files are scattered across the Internet (IEEE TC/OS Newsletter on FTP.CSE.UCSC.EDU

[2]Because the volume of the messages makes it impractical for a single system to archive more than a handful of mailing lists, these archives are scattered across many sites. To the user, however, it looks like a single archive broken down by mailing list.

```
Script started on Wed Jan 29 21:02:50 1992
% cd /
% ls
afs        documents info    mailing-lists  papers    releases
databases  guest     lib     newsgroups     projects  sites
% cd papers
% ls
authors           conferences       subjects
bibliographies    journals          technical-reports
% cd technical-reports
% ls
Berkeley    IAState       OregonSt     UCalgary   UWashington
BostonU     MIT           Purdue       UColorado  Virginia
Chorus      NYU           Rochester    UFlorida   WashingtonU
Columbia    NatInstHealth Toronto      UKentucky
Digital     OregonGrad    UCSantaCruz  UMichigan
% ls UCSantaCruz
crl
% ls UCSantaCruz/crl
ABSTRACTS.1988-89           ucsc-crl-91-01.ps.Z
ABSTRACTS.1990              ucsc-crl-91-02.part1.ps.Z
ABSTRACTS.1991              ucsc-crl-91-02.part2.ps.Z
ABSTRACTS.1992              ucsc-crl-91-02.ps.Z
INDEX                       ucsc-crl-91-03.ps.Z
...
% ls UWashington
cs   cse
%
% ls UWashington/cs
1991           INDEX          PRE-1991
1992           OVERALL-INDEX  README
% cd /papers
% ls
authors           conferences       subjects
bibliographies    journals          technical-reports
% ls journals
acm-sigcomm-ccr  ieee-tcos-nl
% ls journals/ieee-tcos-nl
cfp      v5n1     v5n2     v5n3     v5n4
% ls journals/acm-sigcomm-ccr
app.pps  apr90    jan89    jan91    jul90    oct89
apr89    apr91    jan90    jul89    oct88
% vls journals
   acm-sigcomm-ccr       NNSC.NSF.NET     /usr/ftp/CCR
   ieee-tcos-nl          FTP.CSE.UCSC.EDU /home/ftp/pub/tcos
%
script done on Wed Jan 29 21:06:53 1992
```

Figure 7.1: Sample session

and Computer Communications Review on NNSC.NSF.NET.) Though not shown in the example, papers are also organized by author and subject in other directories from the same virtual system.

Figure 7.2 show a simplified diagram of the virtual system; only a few of the links are actually shown. The figure shows how an online copy of a paper describing the Kerberos authentication system, represented by the black dot, can be found through more than one path: as an MIT technical report (/papers/tech-reports/MIT/authentication.ps), under the directory for the author (/papers/authors/neuman/kerberos.ps), under its subject (/papers/subjects/security/kerberos.ps), and though archie (/match/kerb/kerberos.ps). The directory /match/kerb is created by a filter and is described in greater detail in Subsection 7.2.2.

While it might at first appear that some of the organizational methods employed in this example could be implemented using symbolic links in traditional systems, it is important to keep in mind that symbolic links would break as objects moved, that each link would have to be put in place individually, and that in a system as large as the Internet it would be almost impossible to establish consensus on which directories should appear near the root of the hierarchy, and what files should appear in each subdirectory.

It is also important to note that this example shows only part of the information available through Prospero, and that it shows a typical way that the information is organized. Individuals can, and have, organized their own virtual systems differently. One user, David Cohn, has created a directory named *Nnets* in his own virtual system for the purpose of organizing information about neural networks. That directory has subdirectories for papers, source code, and sample data sets. The papers directory is further broken down by subtopic, and Cohn has added references to papers for each subtopic. Though the directory was organized as part of his own virtual system, links to his Papers directory from other virtual systems allow other users to make use of the data he has organized.

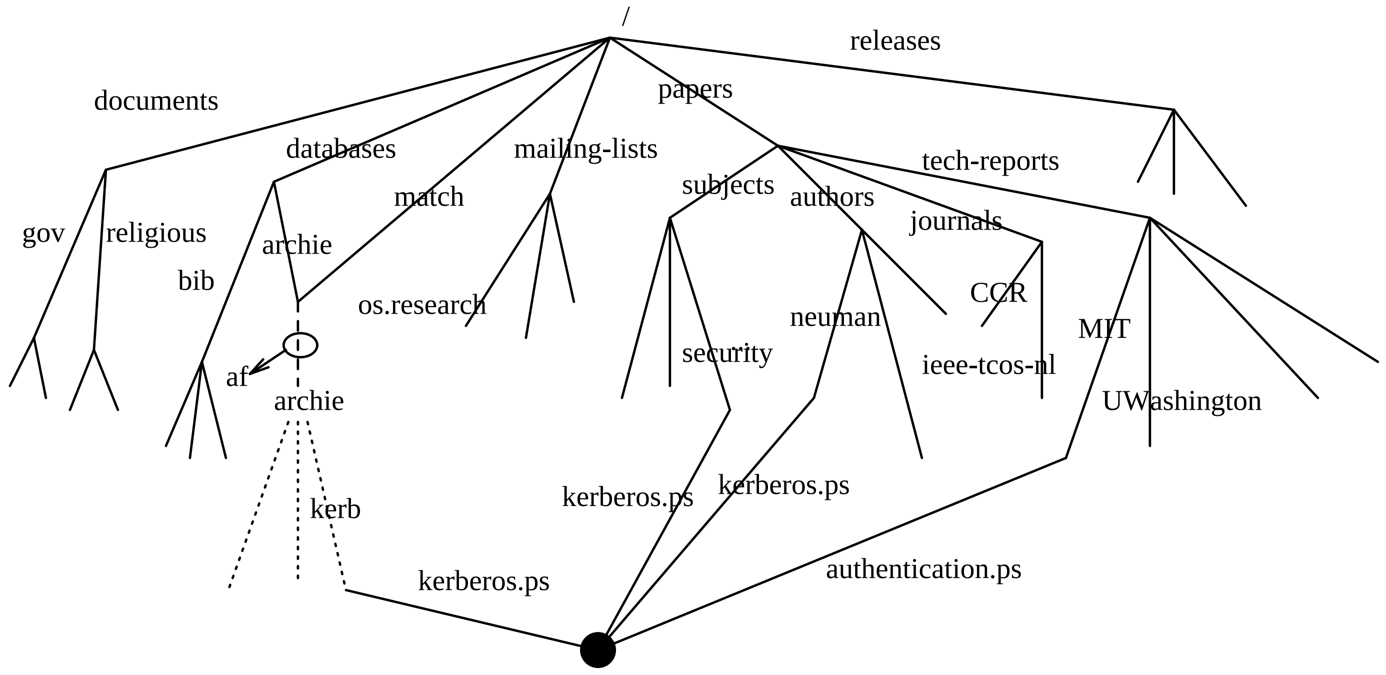


Figure 7.2: A typical virtual system

Figure 7.3 shows a user browsing through Cohn's Nnets directory. As in previous examples, it is important to remember that the files shown are scattered across multiple Internet archive sites, but that it appears as a single system to the user.

7.2.2 Archie

Some of the most frequently used[3] directories are maintained by the archie group at McGill University [Emtage & Deutsch 92]. The archie database [Emtage 91] indexes files from directories at major Internet FTP sites according to the last components of their file names.

Information from the archie database is accessed through a filter (the **af** filter in Figure 7.2). For performance, the filter executes on the Prospero server, where it has direct access to the database, instead of on the client[4]. The directories visible through the filter correspond to the results of queries to the archie database.

For example, the directory *kerb* contains references to files available by Anonymous FTP whose names include the string "kerb". Among the matches would be papers related to the Kerberos authentication system. The contents of each subdirectory are equivalent to what would result from running the Unix `find` command with appropriate arguments over all the major archive sites on the Internet (if it were even possible to do so). The subdirectories do not exist individually, but are instead created by the filter when referenced.

The information in the archie database can be used to simulate the directory structure for the Internet archive sites that are tracked. The archie server supports a second filter that uses this ability to allow Prospero users to navigate through the file hierarchies of

[3]Almost 80 percent of the queries recorded in Subsection 7.1 were to directories maintained by archie. The statistics are skewed, however, by the fact that I had access to the logfiles from the two most prominent archie servers, whereas I was only able to obtain a few logfiles from other Prospero servers.

[4]At present, filters that execute on the Prospero server must be built into the server whereas filters that execute on the client can be dynamically linked during name resolution.

```
Script started on Sun Mar  1 18:13:38 1992
% cd Nnets
% ls
Data          MailingLists  Papers        Sources
% ls Data
CORRESPONDENTS       echocardiogram       molecular-biology
DATE-RECEIVED        flags                mushroom
DOC-REQUIREMENTS     function-finding     othello
Index                glass                pima-indians-diabetes
OVERVIEW             hayes-roth           primary-tumor
TRANSACTIONS         heart-disease        prodigy
access-lists         hepatitis            shuttle-landing-control
annealing            image                soybean
audiology            iris                 spectrometer
autos                kinship              thyroid-disease
breast-cancer        labor-negotiations   tic-tac-toe
bridges              led-display-creator  unavailable
chess                lenses               undocumented
contacts             letter-recognition   university
cpu-performance      liver-disorders      voting-records
dgp-2                logic-theorist       waveform
donations            lymphography         zoo
ebl                  mechanical-analysis
% ls Sources
Biological           backprop.lisp        quickprop1.lisp
Executables          cascor1.c            rcc1.c
MS-DOS               cascor1.lisp         rcc1.lisp
NetTools.tar.Z       hst.README           rcs_v4.2.tar.Z
NeurDS031.tar.Z      hst.tar.Z            vowel.c
PlaNet5.6.tar.Z      mactivation.3.3.sit
am5.tar.Z            quickprop1.c
% ls Papers
Algorithms      CogSci          Implementations  Speech
Applications    Complexity      KnowledgeBase    Topology
Architectures   Control         Misc             Vision
Associative     Generalization  Prediction
Biology         Genetic         Recurrent
% ls Papers/Vision
hinton.handwriting.ps.Z      mozer.segment.ps.Z
maclennan.gabor.ps.Z         zeidenberg.containment.ps.Z
marshall.steering.ps.Z
%
script done on Sun Mar  1 18:16:28 1992
```

Figure 7.3: Sample session

hosts that are not themselves running a Prospero server. Users can then move about the surrogate file hierarchy as if it were available from the host directly, except that the surrogate hierarchy reflects the hierarchy of the host at the time of the last archie database update.

The information from the archie database is not used in isolation, but can be combined with other information available through Prospero. For example, the *Data* directory in the neural-net hierarchy (described in Subsection 7.2.1) includes references from a directory obtained through archie. When the archie database is updated, changes to the archie maintained directory also become visible in the *Data* directory.

The use of archie through Prospero has been so successful that the archie group has adopted Prospero as the preferred method for remote access to the archie database. Work is now underway to make other databases available through Prospero. For example, software releases and other files are often announced on the comp.archives mailing list. By making a database of these messages available through Prospero (in a manner similar to that for the archie database) users will see references to the announced files organized according to keywords from the full text of the announcement.

7.2.3 The Australian Academic and Research Network

The Australian Academic and Research Network (AARNet) Archive Working Group has been a major user of Prospero and has adopted Prospero as part of its plan for archiving in Australia [Cliffe 91].

The Australian Internet is well connected internally, but the connection to the U.S. is over a low speed (512 Kbit) line. Unfortunately, this connection is often used to retrieve the same large files over and over by different Australian users. The traditional solution to this problem was to create an archive site in Australia that maintained a "mirror" of the files on the few most prominent archive sites in the US. Software ran nightly that found changes at the original site, and retrieved new copies, keeping the local copies up-to-date. Problems with this approach are that disk space limits the number of sites that can be mirrored, mirrored files are retrieved whether or not they will be retrieved by local users, and a mirrored copy could be out of date by as much as a day (which presents problems when a new version of a file is announced).

To address these problems, Steve Cliffe has added Prospero support to an FTP server run by the Archive Working Group. As well as making files available from the physical file system, the modified FTP server makes files available from a virtual file system. The virtual file system includes links to the mirrored archive sites, either directly if the site runs Prospero, or to the surrogate directory maintained by archie if it doesn't. Because the files are not copied en masse it is possible to mirror an almost unlimited number of sites. Further, for those sites that run Prospero, the mirror is always up-to-date; as soon as a file changes on the original site, that fact is reflected by the mirror. Finally, only those files actually requested by users are retrieved, reducing network load.

As an added benefit, the modified FTP server provides the benefits of Prospero to users who have not installed it on their systems. Through the FTP server, users can browse through directories in other virtual systems, and selected directories have been included as part of the "ftp" virtual system. This allows users to look for files as they have been organized through Prospero (e.g., as described elsewhere in this chapter), not just as organized by the mirrored archive sites.

When a retrieval request is received, the FTP server locates the file using Prospero and checks to see if a copy of the file is available locally. Using Prospero to check the last modified time of the authoritative copy, the FTP server checks that the local copy is current. If a current copy does not exist locally, the server retrieves and caches a copy of the file. The local copy is then returned to the client.

While it might at first appear that Prospero is being used in Australia primarily for caching, a topic that is not the subject of this dissertation, Prospero is in fact being used exactly as intended. The concept of a mirror has been around for a while, it is well understood, and users are comfortable with it. Prospero allows AARNet to create a virtual mirror, organizing files in a way that is familiar to them, yet guaranteeing that the virtual directories automatically track the originals.

In addition to its use through FTP, Prospero is used directly. The default view for new virtual systems in Australia is a customized view of what is seen in the U.S., but for resources that are available locally (e.g., the local archie server), the links to remote instances have been replaced by links to the local ones. The Australian view automatically tracks the U.S view. If a request is made for a resource that is known to be available locally, a reference to the local copy is returned. If the resource is not available locally, the request is forwarded to the U.S. servers and the non-customized reference is returned.

7.2.4 User Developed Software

It says a lot about a system when others are willing to devote time and resources to developing software that depends on it. Several individuals and groups have done so. AARNet is one example. In addition to the FTP server just described, Steve Cliffe has also contributed a replacement for the directory listing command which will recursively list the directories in a virtual system, stopping the recursion when a cycle is detected.

Other individuals have written or ported programs that provide alternative user interfaces to the information available through Prospero. A standalone client is available that queries the Archie database through Prospero and displays the storage site, filename, and attributes of the files found. Though originally written by myself, Brendan Kehoe of Widener University has improved it and ported it to numerous operating systems. In addition to most varieties of Unix, the standalone client now runs (using Prospero) on MS-DOS and VMS.

Another example is a window-based file browser built by George Ferguson at the University of Rochester. The browser was built on top of Prospero primarily to access the archie database, but it is being extended to support browsing through other parts of the Prospero naming network.

7.3 The Bottom Line

Prospero has been successfully used to organize information available from Internet archive sites. Individuals have used it to to organize information on many topics, and network service providers are starting to use it to organize information of interest to their users. I have also been contacted by two universities that are interested in using Prospero to implement campus-wide, network-transparent information services.

Experience has shown that Prospero helps users find the information they are looking for. In the past, users looking for files on particular topics would send messages to appropriate (and sometimes inappropriate) mailing lists asking if any one knows where to look. Many users still send such messages, but others are able to find the answer using Prospero, reducing the frequency of such messages. I have even seen people respond to such messages with the results of Prospero queries.

The bottom line is that Prospero helps users organize information and find the information they're looking for. Analysis of server logs shows that most individuals who use Prospero once, use it again.

Chapter 8

Related Problems

The problem of finding and organizing information is a problem that spans several areas of research. I have approached the problem from the systems perspective with a particular emphasis on the effects of scale. The problem could also be approached from several other perspectives. In this chapter I discuss the relationship of the Virtual System Model to other work both in the systems and non-systems areas.

8.1 Directory Services

Distributed directory services are used to organize information in large systems. Chapter 3 discussed the problems with global directory services such as the Internet Domain Naming System [Mockapetris 87], X.500 [CCITT 88], and DEC's Global Naming System [Lampson 85]. While these systems are useful for organizing information whose structure conforms to the administrative relationship between different parts of the system, they are less useful when the information is structured in other ways. The reason is that the same name space is visible to everyone, and users are limited to modifying their own parts of the name space. The result is that related information is scattered across the name space and difficult to locate without knowing who created it.

Attribute-based naming, as supported by Profile [Peterson 88], presents an alternative to the global hierarchical name services just described. In attribute-based naming, objects are named by a collection of attributes. When naming an object, attributes may be omitted, and only enough attributes to uniquely identify the object must be provided. An advantage of attribute-based naming over hierarchical name spaces is that the information appears to be organized in multiple ways: users can find objects even if some information, e.g., the owner of the object, is not known. However, the attributes that users will use in their search must be registered ahead of time.

Unfortunately, attribute-based naming in its pure form does not scale well. It is difficult to distribute the database of attributes across a large number of servers. Without a way to direct a query to the right server, queries must be sent to all servers, an operation that doesn't scale. Profile [Peterson 88] restricts the set of name servers that are queried and relies on cross-references to direct queries to servers that were not included in the original set. This approach requires that the right cross-references be in place before the query is made, negating one of the advantages of attribute-based systems, i.e., that the links corresponding to a particular query need not be in place ahead of time.

8.2 File Systems

There is a huge amount of information that is stored in conventional file systems and it would make sense to organize it using the mechanisms already supported by those file systems. Chapter 3 discussed some of the shortcomings of conventional distributed file systems like the Andrew File System [Howard et al. 88], Coda [Satyanarayanan 90, Kistler & Satyanarayanan 91], Locus [Walker et al. 83], Sprite [Ousterhout et al. 88], and Echo [Hisgen et al. 89]. Because these systems support a single name space, they suffer from the same organizational problem described above: it is difficult to organize information whose logical structure doesn't follow the administrative structure of the system.

By using aliases (symbolic links), it is possible to create a directory with links to related objects scattered across the name space, but such links will cease to work if the primary name for the object changes. Further, the name of the directory that contains the links will still be constrained by administrative concerns, making it difficult for users to find.

Conventional file systems provide only limited support for customization. Users can customize their own directories and subdirectories, and they can include links to other parts of the global name space. However, such customized directories will have different names than the parts of the global name space that they replace, meaning that users have to remember which parts of the name space have been customized.

Some recent systems including Tilde [Droms 86, Comer et al. 90], QuickSilver [Cabrera & Wyllie 88], Plan 9 [Presotto et al. 91], and Amoeba [Tanenbaum et al. 90, van Renesse 89] allow users to customize all parts of their name spaces. As mentioned in Chapter 3 however, these systems fail to address the lack of name transparency that results when multiple name spaces are supported, a problem that is addressed in the Virtual System Model by supporting closure.

Existing file systems do not provide adequate tools to help users create customized views in terms of directories that already exist. Without such tools, users can only customize their name spaces by creating new directories which shadow original directories item by item. To do so is tedious, and the customized directory would need to be updated whenever the original directory changed. In the Virtual System Model the filter and the union link allow directories to be created as functions of other directories and the resulting directories automatically track the directories from which they are derived.

Semantic file systems [Gifford et al. 91] provide an alternative method for finding files of interest from a collection of file servers. A semantic file system automatically extracts attributes for the files it contains. Users search for files by specifying attributes which are looked up in the resulting database. It is easier to find information of interest in a semantic file system because, as in the Virtual System Model, users can look for information in different ways depending on what they know about the object they are looking for.

Just as a filter in the Virtual System Model allows the result of a computation (e.g., a database query) to be presented to the user in a virtual directory, the semantic file system described by [Gifford et al. 91] does the same. An important difference is that the mapping between file names and queries in a semantic file system is specified by the implementation of the file server, whereas in the Virtual System Model, the mapping is specified by a filter which can be written (or selected) by the user performing the search or creating a derived view.

Unlike the Virtual System Model, which relies on users or services to create the appropriate links, a semantic file system automatically generates the attributes used for searches[1]. However, a semantic file system has the same problem inherent in other attribute based naming mechanisms: that it is difficult to distribute the database of attributes across a large number of servers. When used together, the Virtual System Model and semantic file systems could be very powerful. The databases maintained by semantic file systems could be accessible through filters which could perform any desired pre- or post-processing, and other features of the Virtual System Model could be used to impose a structure that directs queries to the appropriate file servers.

8.3 Databases

Databases have been used to organize large collections of information for some time. Early examples include commercial database services such as Lexis and Westlaw which support full text searches of legal rulings. More recently, the Wide Area Information Service (WAIS) [Kahle & Medlar 91] has provided a common protocol to access multiple

[1]Attributes are generated by *transducers*, type-specific programs that read files and produce attributes. Users can write their own transducers, and the transducers that are applied can depend on the position of a file in the file hierarchy. It is the user or group with administrative control over a file that determines which transducers are applied, not necessarily the user doing the searching.

full text databases over the Internet. Attribute-based naming and semantic file systems provide other examples of the use of databases to organize information.

These systems work by computing an index over the information in the database. In a large system, the amount of data to be indexed, the frequency of updates, and the need to cross administrative boundaries would preclude the use of a single index. Distributed indexing [Danzig et al. 91] is a proposed approach for generating and maintaining multiple indices. Each index would contain information on a particular topic, separate from the data being indexed. To use multiple indices for organizing information in a large system would still require the development of a mechanism to direct queries to the appropriate index. The Virtual System Model would fill this role nicely.

In fact, although it doesn't completely conform to existing database models, the naming network of the Virtual System Model can be thought of as a database. It could be implemented using a database based on the network model [Date 81]. Each object would be represented by a record having unique identifying information that allows it to be referenced or uniquely specified as the starting node (root) of a name space. Names would be resolved by following the named (or unnamed) links that connect the directory records with the records representing its subdirectories or other referenced objects.

Even using the network model, there would be some features of the directory service that could not be easily supported. One of these features is views. Although database models support multiple views, there are some important differences between database views and views as defined by the Virtual System Model.

The primary functions of a database view are as an abstraction mechanism that hides the conceptual organization of the information in a database from the application and as a protection mechanism that prevents access to information that is to be protected [Wiederhold 86]. In the Virtual System Model, views are used to organize the available information.

A second important difference between database views and views in the Virtual System Model is that database views are specified external to the database and are a shorthand for the result of applying database operations to one or more databases. Queries can be made to these views as if they were separate databases, and the query is translated to the corresponding query on the databases on which the view is based[2].

In the Virtual System Model views are an integral part of the naming network. The functions that help define views are associated with the links in the naming network and they affect the way that one moves through the network. The observed view is not specified outside of the network. Instead, it is determined by the links that are followed when resolving a name within the network.

Distributed databases have received a great deal of attention in recent years and it might be possible to apply that work to the problem of building a single database within which all information could be organized. A distributed database is a collection of data that belong logically to the same system but are spread across the sites of a computer network [Ceri & Pelagatti 84]. There has been much work on merging dissimilar databases (often across multiple systems) into a single database [Litwin & Abdellatif 86]. Much of this work has concentrated on schema integration [Batini et al. 86], determining what the underlying databases have in common and deriving a new model (schema) that integrates the data in each of the underlying databases.

Schema integration is only useful for tying together logically related information. Many pairs of databases are not related in any useful manner. Consider, for example, a database containing the average yearly rainfall for all cities in the State of Washington, and a database with information on articles published in *Communications of the ACM*. When there is no useful relationship between the data in two databases, they should be treated as separate.

When looking for information stored in a database (or through a view defined on one or more databases), it must be possible to identify the database or view to be queried. This ability is also necessary when defining views on one or more databases. Databases may be identified using higher level databases, or through traditional naming techniques.

[2]It is sometimes possible to cache the intermediate results of applying the view, in which case, the query may be applied directly to the cached results

In a federated database architecture [Heimbigner & McLeod 85] autonomous sites export definitions of the databases that are available to users at other sites on the network. A definition is exported by adding an entry to a database that is shared by all members of the federation. This meta-database imposes a uniform global name space on the federation. As the federation grows it will become harder to reach agreement on how entries in the database should be organized. It will also become necessary to distribute the database over a greater number of sites (for reliability and performance) but the scalability of existing distributed database systems is limited.

When traditional naming techniques are used to name the available databases they usually impose a global hierarchical name space with the site at its first level, and a local name for the database further down. The same problems that were discussed in Chapter 3 apply to the naming of databases. Database systems could benefit from applying the Virtual System Model to the naming of individual databases.

There are some important differences between a name service and a database. The primary difference is that the names used to query a name server are much less constrained than the keys usually used in database systems. Though names are often meaningful, they may be meaningful only to a particular user, and often names used in the same directory will come from completely different domains. This can be useful; the name for an object will often capture the critical distinguishing feature for that object, though the critical distinguishing feature might be different for different objects in the same directory. For objects that the user has named, or that have been previously seen, it is often easier to identify and remember the object by name than by a set of attributes that uniquely identifies the object. The same may be true for objects that the user has not yet seen, if the information has been organized appropriately.

8.4 Computer Supported Cooperative Work

The field of computer supported cooperative work is concerned with providing tools that make it easier for users to work together. In order to collaborate, users must be able to share objects, and much of the work in CSCW is concerned with the way that information is shared within and across organizations. This section briefly discusses the relationship between CSCW and the Virtual System Model.

In order to collaborate, users must be able to understand one another. When working together through a computer, users refer to the objects stored in the computer using names. It must be possible for the recipient of a messages to resolve names that are mentioned within the message. The Virtual System Model makes this possible with closure.

When users work together there might also be a shared workspace. This is the mechanism by which changes made by one user become visible to others. With a name space based on the Virtual System Model, the shared workspace can be represented by a directory shared by the users working on a project. The individual files that are being worked on could be stored across multiple sites, but the view seen by the collaborating users would be the same as if they were using the same system.

Users are members of multiple groups. Systems should allow users to integrate the workspaces of the groups in which they participate. The tools provided by the Virtual System Model allow them to do so.

8.5 Hypertext

Hypertext has gained widespread interest as an approach for organizing the information in computer systems. This section discusses the relationship between hypertext and the Virtual System Model.

Hypertext [Nelson 65, Conklin 87] is the combination of text with links to other text. These links represent cross references which may be followed when searching for information, and they allow the interconnection of documents in ways that are more general than supported by traditional file systems. The term hypertext also refers to hypermedia, a generalization where the objects organized may be other than text.

Hypertext is useful for finding information because it can represent a greater degree of interconnection between objects. An object can be reached through more than one path, and links to an object can be created wherever the object is mentioned. This makes a linked object easily referenceable at those points where the user is most likely to reference it. Users find information by moving from object to object, selecting the link to be followed at each step.

While useful for organizing information and for browsing documents, there are several characteristics of hypertext that make it unsuitable for use as the only mechanism for identifying objects. One such characteristic is that displaying each object along the path followed to reach that object is cumbersome when one knows ahead of time which object is to be accessed. In such situations it should be possible to name the object directly. In fact, recognizing this need, many hypertext systems support names as an alternative method for identifying objects. Names are also important if one is to refer to an object through any mechanism other than the hypertext system itself, for example in person, over the phone, or through an online application such as electronic mail which might not have been integrated with the hypertext system.

Another problem with many hypertext systems is that it is easy to get lost. The sequence of links to be traversed to identify an object depends on one's current position in the hypertext graph. As one moves through the graph it is possible to lose track of one's position relative to other objects of interest, and it isn't always a simple matter to find one's way back to intermediate nodes that were previously visited.

A directory service based on the Virtual System Model supports the same degree of interconnection that exists in hypertext systems. In the Virtual System Model, objects have multiple names and can appear in multiple directories, allowing them to be organized in multiple ways. Cross references between objects can be represented as attributes. The primary difference between the Virtual System Model and hypertext systems is that when using the Virtual System Model, links can have names, and the concatenation of the names on the links along any path to an object can be used by any application as a name for that object. Additionally, the Virtual System Model provides tools that allow users to customize the name space. Links in hypertext documents can originate at any point within the body of a document, whereas using the Virtual System Model links and cross references can only be associated with the document as a whole; this limitation is relatively easy to work around, however.

Most hypertext systems run as a single application on a single system, but there has been recent work on supporting hypertext links across applications and over a network [Yankelovich et al. 88, Kacmar & Legget 91, Pearl 89]. Most existing hypertext systems that work across application boundaries make use of a link service [Pearl 89] or database [Riley 89] to maintain information about the links. Existing link services are not capable of tying together information on a global scale, though there is a great deal of interest in building link servers that can. In fact, several projects, including Bootstrap [Engelbart 90] and World Wide Web [Berners-Lee et al. 92], are attempting to build open hypertext systems that tie together information distributed across the world; these projects are still in their early stages.

A distributed link service could be easily built on top of Prospero (or any system that satisfies the requirements of the Virtual System Model). The links that connect documents would be implemented as named cross references associated with files. The names of the cross references would encode information about the source of the link, perhaps in the form of an offset into the file, or in the form of a token embedded in the file itself. The target of the link would be an entire file, or by defining a new address type, the link could reference a region within a file.

8.6 The Resource Discovery Project

The resource discovery project at the University of Colorado [Schwartz 91] has examined mechanisms for the discovery of resources on large networks. Most of the work has concentrated on ways to use the existing information sources that already exist on the network, often tying together multiple services. For example, a white pages tool was implemented that supports locating individuals over the network. It starts its search by looking for the organization name in a database built from network news articles. It then uses the Internet Domain Naming System to find hosts to look for and finally the finger protocol to search for the individual.

Other approaches include user discovery agents [Schwartz 89] which accept queries from users and use the information provided in the query to find objects in which the user has an interest. The information needed to direct a query to the appropriate agent evolves over time. A query is directed to the nearest agent, and agents learn how to direct queries based on the results of previous queries. A potential problem with this approach is that it gives the agents too much of the responsibility for building the resource discovery graph and for developing the heuristics for navigating through it. Some techniques have been suggested for making the agents capable in this regard, but I do not believe that these techniques will scale.

Another approach has been applied to finding information available from Internet archive sites. The tools provided look for information in databases that are maintained by individual archive sites. This information is augmented with data that has been cached as users search for information. They are now looking at some of the same mechanisms already supported by the Virtual System Model to allow users and groups of users to construct views that organize the information that is available.

Many of the approaches taken by the resource discovery project work because of the potentially rich interconnection between resources. The greater the interconnection, the fewer the steps that are required to find a resource. There is much information which might be useful in finding objects, but which to date could only be obtained by external means (such as asking the appropriate person). Using the Virtual System Model this information can be explicitly included in the directory service in the form of links, cross-references, attributes, and filters. Tools for resource discovery could benefit from the use of this information in directing searches.

8.7 Operating Systems

The goal of allowing users to construct a system by selecting components that are available from the network is a goal that is shared by Plan 9 [Presotto et al. 91]. A key difference between Plan 9 and the Virtual System Model is that Plan 9 addresses the problems of combining the components, not of finding them.

A system view in Plan 9 is not persistent. Instead, it is constructed by a process (often using a configuration file) and only lives as long as the processes that uses it. To construct a system view, the user must specify a globally unique name for each component, introducing the problems associated with global naming.

Plan 9 does not address the problem of how users identify the components that they want to include in their system view. The Virtual System Model is concerned primarily with the mechanisms needed to organize and identify the components of interest and relies on system provided access methods to actually use them.

Chapter 9

Conclusion

Naming is a critical service in distributed systems. Users employ names to identify the objects accessible from within the system: files, services, processors, and other users. The name service translates the names of objects to the information needed by the computer to access those objects.

The size of distributed systems is growing. As the amount of information that is available grows, so does the number of objects that need to be named. Existing techniques for naming in distributed systems have evolved from the techniques used on centralized systems. These techniques are not sufficient for organizing the large amount of information that is becoming available.

In this dissertation I have argued that existing models of naming are not sufficient for organizing information in extremely large systems. In existing systems, the names that can be assigned to objects are constrained by the object's location or the identity of the individual assigning the name. This limits the ability of users to find an object when they don't know where the object is stored, or who owns it. The assignment of names should not be so constrained. Additionally, users need the ability to customize their views of the system, eliminating those parts that are not of interest.

9.1 Contributions

The contributions of this dissertation are:

- The recognition that users need the ability to view large systems in different ways and that a uniform global name space is not appropriate for user-level naming in extremely large systems.

- The recognition that any system that supports multiple views or multiple name spaces, whether as an integral part of the naming mechanism or through ad hoc means, must also address the problems that result from the lack of name transparency.

- The presentation of the Virtual System Model, a new model for naming that better supports customization. The Virtual System Model provides powerful tools which help users define their own views of the system, either directly, or in terms of other views. The Virtual System Model addresses the lack of name transparency by supporting closure.

- A description of the manner in which the Virtual System Model can be applied to the organization of computer systems, both in the naming and organization of files, and in its application to other services including authentication, authorization, and the selection of processors, services, and applications.

- The design and implementation of Prospero, a prototype file system based on the Virtual System Model. Prospero supports the creation of multiple virtual file systems from the files that are available over the Internet.

- The dissemination and support of Prospero as an experiment to validate the ideas embodied in the Virtual System Model. The widespread use of the prototype to organize and identify information of interest demonstrates the usefulness of the model.

Like much recent work on large distributed systems, this dissertation addresses problems of scale. It differs from other work in the area by concentrating on how scale affects the user, an aspect of the problem that has been largely ignored until now.

9.2 Future Work

Evaluation of the prototype demonstrates the usefulness of the Virtual System Model. The widespread use of the prototype indicates a need for the functionality it provides. The success of the prototype warrants continued work in the area. This work falls into three categories: enhancements to Prospero, further experiments with the Virtual System Model, and new research.

The first category, enhancements to Prospero, consists of improvements that will allow Prospero to gain even wider use. With wider use, more information will become available through Prospero, allowing a more accurate evaluation of the scalability of the mechanisms it provides. Planned enhancements to Prospero include:

- Making additional services available through Prospero. Work is already underway to make Wide Area Information Service (WAIS) databases available. Other candidates include the comp.archives database, library card catalogs, and mailing list servers.

- Adding support for object types that are not directly accessible, or that are accessible only after considerable delay. The access methods for such objects would provide information on how to access them, whether that involves actions like visiting a library, or submitting retrieval requests.

- Making it possible for programs that have not been linked with the compatibility library to use the Prospero file system.

- Running filters in separate address spaces and with restricted privileges. In the present implementation, a filter runs with the privileges of the individual applying it. Since the individual applying the filter might not even be aware that it is being applied, this presents a security weakness.

- Allowing users to specify the application of a filter as part of a file name. The filter would be specified as one of the components in the file name. The components of the name that follow the filter would be resolved in the virtual directory created by the application of the filter to the directory named in the components that precede the filter.

- Allowing the server to apply an arbitrary filter (when requested by the client). If the filter requires information about objects that are stored on the server, this would be more efficient than running the filter on the client. The filter would run in a separate address space with restricted privileges.

- Adding support for alternative types of filters. Possibilities include filters that return the value of a named link instead of returning the entire directory (selection filters), filters that translate requests for changes to a virtual directory into the corresponding change to the underlying physical directories (update filters), and filters that affect access to the object itself, not just the listing of directories (data filters).

- Adding support for the replication of directories, and implementing alternative replication methods for other types of objects.

- Adding support for the freezing of closures. At present, closures are dynamic. When a change is made to a name space, that change affects the resolution of names by objects bound to the name space through closure. This can present problems for security. The ability of the user to control the name space used by a program is important if a program is to be confined[1] [Lampson 73].

[1]A program is confined when data cannot be written to locations that are not intended by the user.

The second category, further experiments with the Virtual System Model, consists primarily of applying the Virtual System Model to other parts of the system. Chapter 5 described how the features of the Virtual System Model affect other parts of the system, but the ideas have not been tested. Planned experiments include:

- Letting the active virtual system specify the set of processors on which a user's processes can run.
- Allowing the virtual system closed with a program to restrict the set of processors on which the program can run.
- Using filters to select the instance of a service to be used based on possibly changing characteristics such as price, load, and locality.
- Using the virtual system closed with an object to specify the authentication and authorization mechanisms to be employed by principals wishing to access the object.
- Allowing users to specify user names and group names from their customized name space when modifying access control lists, and allowing them to see customized names when they view such lists.
- Using closure to restrict the files that a process can read and write. By building a virtual system with limited size, by protecting the virtual system so that it can't be modified from within, and by restricting the resolution of names to those in the restricted virtual system, a process can be run in a protected environment.

Work is underway on the first two items above: DARPA funded research at the University of Southern California's Information Sciences Institute is applying the concepts of the Virtual System Model to resource management on multiprocessor systems.

The third category, new research, consists of solving problems that arise because of the flexibility of the Virtual System Model and the greater size of the systems that can be built using the model. It also includes solving problems for which new approaches are possible when built upon the infrastructure supported by the model. More work is needed in the following areas:

- Inverted naming. The Virtual System Model allows objects to have many names. Given an object, it should be possible for users to determine some of the existing names for the object relative to the active virtual system. This ability is important when browsing or when a reference to an object is received from another user because it allows one to determine whether the object is already known. Inverted naming could be implemented using breadth first search, but faster methods are needed.

- Garbage collection in large systems. The mechanism described in Chapter 6 will work for very large systems, but it won't scale indefinitely. Prospero provides an ideal test bed for experimenting with new algorithms.

- The view maintenance problem. The results of applying filters or expanding a union link can be cached, but a method is needed to keep the cached view consistent with the directories on which it is based.

- The view update problem. When the user makes a change, a method is needed to decide whether the change is a customization that should affect only the user's view, or whether it is a modification that should be visible to everyone. If the change is specified in terms of a derived view, it is also necessary to determine how that change should be reflected in the physical directories that underlie the view. The update filter might solve part of this problem.

- Tools for finding information. The Virtual System Model provides tools for organizing information. It is easier to find things that have been organized using the tools provided, but it is the user's responsibility to navigate through the naming network in search of the desired information. The Virtual System Model provides the infrastructure on top of which tools for finding information can be implemented.

9.3 Final Remarks

The Virtual System Model provides a powerful framework within which information can be organized. The Prospero prototype makes that framework available for organizing information on the Internet. By themselves, neither the model nor the prototype help users find information of interest. Their contributions rest in encouraging and enabling users to organize information in ways that make it easier to find things.

Professional societies, libraries, governments, commercial indexing services, and others will play important roles in organizing the information available from future systems. The Virtual System Model allows such service providers to build upon each other's work, eliminating duplicated effort, and it allows users to construct views of the information provided by these services which better meet their own requirements. The real contribution of this work will be measured by the extent to which the model is adopted by service providers and used to organize systems. The early signs are extremely positive in this regard.

Bibliography

[Batini et al. 86] C. Batini, M. Lenzerini, and S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, December 1986.

[Berners-Lee et al. 92] T. Berners-Lee, R. Cailliau, J.-F. Groff, and B. Pollermann. World-wide web: The information universe. *Electronic Networking: Research, Applications and Policy*, 2(1), Spring 1992.

[Bershad & Pinkerton 88] B. N. Bershad and C. B. Pinkerton. Watchdogs: Extending the Unix file system. *Computing Systems*, 1(2):169–188, Spring 1988.

[Birrell et al. 86] A. D. Birrell, B. W. Lampson, R. M. Needham, and M. D. Schroeder. A global authentication service without global trust. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 223–230, April 1986.

[Black 91] A. P. Black. Personal communication, July 1991.

[Brownbridge et al. 82] D. Brownbridge, L. Marshall, and B. Randell. The Newcastle connection or UNIXes of the world unite. *Software: Practice and Experience*, 12:1147–1162, 1982.

[Cabrera & Wyllie 88] L.-F. Cabrera and J. Wyllie. QuickSilver distributed file services: An architecture for horizontal growth. In *Proceedings of the 2nd IEEE Conference on Computer Workstations*, pages 23–27, March 1988.

[CCITT 88] CCITT. Recommendation X.500: The Directory, December 1988.

[Ceri & Pelagatti 84] S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems.* Computer Science Series. McGraw-Hill, 1984.

[Cliffe 91] S. Cliffe. Archiving within Australia. Department of Computer Science, University of Wollongong, Australia, April 1991.

[Comer & Peterson 89] D. E. Comer and L. L. Peterson. Understanding naming in distributed systems. *Distributed Computing*, 3(2):51–60, 1989.

[Comer et al. 90] D. Comer, R. E. Droms, and T. P. Murtagh. An experimental implementation of the Tilde naming system. *Computing Systems*, 4(3):487–515, Fall 1990.

[Conklin 87] J. Conklin. Hypertext: An introduction and survey. *IEEE Computer*, 20(9), September 1987.

[Daley & Neumann 65] R. Daley and P. G. Neumann. A general purpose file system for secondary storage. In *Proceedings of the AFIPS Fall Joint Computer Conference*, pages 213–229, 1965.

[Danzig et al. 91] P. B. Danzig, J. Ahn, J. Noll, and K. Obraczka. Distributed indexing: A scalable mechanism for distributed information retrieval. In *Proceedings of the ACM SIGIR Conference on Research and Development in Information Retrieval*, October 1991.

[Date 81] C. J. Date. *An Introduction to Database Systems.* The Systems Programming Series. Addison-Wesley, 1981.

[Dig 80] Digital Equipment Corporation, Maynard, Massachusetts. *TOPS-20 User's Guide (Version 4)*, January 1980.

[Dig 88] Digital Equipment Corporation, Maynard, Massachusetts. *VMS General User's Manual (Version 5)*, April 1988.

[Droms 86] R. E. Droms. *Naming of Files in Distributed Systems.* Ph.D. Dissertation, Purdue University, August 1986.

[Emtage & Deutsch 92] A. Emtage and P. Deutsch. archie: An electronic directory service for the Internet. In *Proceedings of the Winter 1992 Usenix Conference*, pages 93–110, January 1992.

[Emtage 91] A. Emtage. archie. M.S. Thesis, McGill University, May 1991.

[Engelbart 90] D. C. Engelbart. Knowledge-domain interoperability and an open hyperdocument system. In *Proceedings of the Conference on Computer Supported Cooperative Work*, pages 143–156, October 1990.

[Fowler 85] R. J. Fowler. *Decentralized Object Finding Using Forwarding Addresses.* Ph.D. Dissertation, University of Washington, December 1985. Department of Computer Science Technical Report 85-12-01.

[Gifford et al. 91] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'Toole Jr. Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 16–25, October 1991.

[Heimbigner & McLeod 85] D. Heimbigner and D. McLeod. A federated architecture for information management. *ACM Transactions on Office Information Systems*, 3(2), July 1985.

[Hisgen et al. 89] A. Hisgen, A. Birrell, T. Mann, M. Schroeder, and G. Swart. Availability and consistency tradeoffs in the Echo distributed file system. In *Proceedings of the 2nd IEEE Workshop on Workstation Operating Systems*, pages 49–54, September 1989.

[Ho & Olsson 90] W. W. Ho and R. A. Olsson. An approach to genuine dynamic linking. Technical Report CSE-90-25, Department of Computer Science and Engineering, University of California at Davis, August 1990.

[Hopcroft & Ullman 79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*, chapter 2. Addison-Wesley, Reading, Massachusetts, 1979.

[Howard et al. 88] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[Kacmar & Legget 91] C. J. Kacmar and J. J. Legget. PROXHY: A process-oriented extensible hypertext architecture. *ACM Transactions on Information Systems*, 9(4):399–419, October 1991.

[Kahle & Medlar 91] B. Kahle and A. Medlar. An information system for corporate users: Wide area information systems. Technical Report TMC-199, Thinking Machines Corporation, April 1991.

[Kistler & Satyanarayanan 91] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 213–225, October 1991.

[Korn & Krell 90] D. G. Korn and E. Krell. A new dimension for the Unix file system. *Software: Practice and Experience*, 20:19–34, June 1990.

[Lampson 73] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973.

[Lampson 85] B. W. Lampson. Designing a global name service. In *Proceedings of the 4th ACM Symposium on Principles of Distributed Computing*, August 1985.

[Lantz et al. 85] K. A. Lantz, J. L. Edighoffer, and B. L. Hitson. Towards a universal directory service. In *Proceedings of the 4th ACM Symposium on Principles of Distributed Computing*, August 1985.

[Lindsay 81] B. Lindsay. Object naming and catalog management for a distributed database manager. In *Proceedings of the 2nd International Conference on Distributed Computing Systems*, pages 31–40, April 1981.

[Litwin & Abdellatif 86] W. Litwin and A. Abdellatif. Multidatabase interoperability. *IEEE Computer*, 19(12):10–18, December 1986.

[Mealy 55] G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[Mockapetris 87] P. Mockapetris. Domain names - concepts and facilities. DARPA Internet RFC 1034, November 1987.

[Moore 64] E. F. Moore, editor. *Sequential Machines: Selected Papers.* Addison-Wesley, Reading, Massachusetts, 1964.

[Mullender & Tanenbaum 86] S. J. Mullender and A. S. Tanenbaum. The design of a capability-based distributed operating system. *The Computer Journal*, 29(4):289–299, 1986.

[Nelson 65] T. H. Nelson. A file structure for the complex, the changing, and the indeterminate. In *Proceedings of the ACM National Conference*, August 1965.

[Neuman 88] B. C. Neuman. Issues of scale in large distributed operating systems. General Examination Report, Department of Computer Science, University of Washington, May 1988.

[Neuman 89a] B. C. Neuman. The need for closure in large distributed systems. *Operating Systems Review*, 23(4):28–30, October 1989.

[Neuman 89b] B. C. Neuman. The Virtual System Model for large distributed operating systems. Technical Report 89-01-07, Department of Computer Science, University of Washington, April 1989.

[Neuman 89c] B. C. Neuman. Workstations and the Virtual System Model. In *Proceedings of the 2nd IEEE Workshop on Workstation Operating Systems*, pages 91–95, September 1989. Also appears in the *Newsletter of the IEEE Technical Committee on Operating Systems*, Volume 3, Number 3, Fall 1989.

[Neuman 90] B. C. Neuman. Managing replicated data within the Virtual System Model. Department of Computer Science and Engineering, University of Washington, May 1990.

[Neuman 92a] B. C. Neuman. Prospero: A tool for organizing Internet resources. *Electronic Networking: Research, Applications and Policy*, 2(1):30–37, Spring 1992.

[Neuman 92b] B. C. Neuman. The Prospero File System: A global file system based on the Virtual System Model. In *Proceedings of the Workshop on File Systems*, May 1992.

[Neuman 92c] B. C. Neuman. Scale in distributed systems. In *Readings in Distributed Computing Systems*. IEEE Computer Society Press, 1992.

[Notkin & Griswold 88] D. Notkin and W. G. Griswold. Extension and software development. In *Proceedings of the 10th International Conference on Software Engineering*, pages 274–283, April 1988.

[Ousterhout et al. 88] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The Sprite network operating system. *Computer*, 21(2):23–35, February 1988.

[Pearl 89] A. Pearl. Sun's link service: A protocol for open linking. In *Proceedings of Hypertext'89*, pages 137–146, November 1989.

[Peterson 88] L. L. Peterson. The Profile naming service. *ACM Transactions on Computer Systems*, 6(4):341–364, November 1988.

[Postel & Reynolds 85] J. B. Postel and J. K. Reynolds. File transfer protocol. DARPA Internet RFC 959, October 1985.

[Postel 80] J. B. Postel. User datagram protocol. DARPA Internet RFC 768, August 1980.

[Presotto et al. 91] D. Presotto, R. Pike, K. Thompson, and H. Trickey. Plan 9: A distributed system. In *Proceedings of Spring 1991 EurOpen*, May 1991.

[Rabin & Scott 59] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research*, 3(2):115–125, 1959.

[Riley 89] V. A. Riley. An interchange format for hypertext systems: The intermedia model. Technical Report 89-6, Institute for Research in Information and Scholarship, Brown University, 1989.

[Ritchie & Thompson 74] D. M. Ritchie and K. Thompson. The Unix time sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[Saltzer 78] J. H. Saltzer. *Operating Systems: an advanced course*, volume 60 of *Lecture Notes in Computer Science*, chapter 3, pages 99–208. Springer-Verlag, 1978.

[Saltzer 82] J. H. Saltzer. On the naming and binding of network destinations. In *Proceedings of the International Symposium on Local Computer Networks*, pages 311–317, April 1982.

[Sandberg et al. 85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network File System. In *Proceedings of the Summer 1985 Usenix Conference*, pages 119–130, June 1985.

[Satyanarayanan 90] M. Satyanarayanan. Scalable, secure, and highly available distributed file access. *IEEE Computer*, 23(5):9–21, May 1990.

[Schwartz 89] M. F. Schwartz. The networked resource discovery project. In *Proceedings of the IFIP XI World Congress*, pages 827–832, August 1989.

[Schwartz 91] M. F. Schwartz. Resource discovery and related research at the University of Colorado. Technical Report CU-CS-508-91, Department of Computer Science University of Colorado, Boulder, January 1991.

[Sollins 85] K. R. Sollins. *Distributed Name Management.* Ph.D. Dissertation, Massachusetts Institute of Technology, February 1985. Laboratory for Computer Science Technical Report 331.

[Steiner et al. 88] J. G. Steiner, B. C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the Winter 1988 Usenix Conference*, pages 191–201, February 1988.

[Stoy & Strachey 72] J. E. Stoy and C. Strachey. OS6: An experimental operating system for a cmall computer. *The Computer Journal*, 15(3):195–203, 1972. Part 2: Input/output and filing system.

[Tanenbaum et al. 90] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. van Rossum. Experience with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):47–63, December 1990.

[Terry 85] D. B. Terry. *Distributed Name Servers: Naming and Caching in Large Distributed Computing Environments.* Ph.D. Dissertation, University of California, Berkeley, March 1985. Computer Science Division Technical Report 85-228.

[Terry et al. 84] D. B. Terry, M. Painter, D. W. Riggle, and S. Zhou. The Berkeley internet domain server. In *Proceedings of the Summer 1984 Usenix Conference*, pages 23–31, June 1984.

[Tichy & Ruan 84] W. F. Tichy and Z. Ruan. Towards a distributed file system. In *Proceedings of the Summer 1984 Usenix Conference*, pages 87–97, June 1984.

[Turing 36] A. M. Turing. On computable numbers with an application to the entscheidungsproblem. *Proceedings of the London Math Society*, 2(42):230–265, 1936.

[van Renesse 89] R. van Renesse. *The Functional Processing Model.* Ph.D. Dissertation, Vrije Universiteit, Amsterdam, 1989.

[Walker et al. 83] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The Locus distributed operating system. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 49–70, October 1983.

[Wiederhold 86] G. Wiederhold. Views, objects, and databases. *IEEE Computer*, 19(12):37–44, December 1986.

[Yankelovich et al. 88] N. Yankelovich, B. J. Haan, N. K. Meyrowitz, and S. M. Drucker. Intermedia: The concept and the construction of a seamless information environment. *IEEE Computer*, 21(1), January 1988.

Appendix A

Prospero User's Manual

The Prospero file system is based on the Virtual System Model. It differs from traditional distributed file systems in several ways. In traditional file systems, the mapping of names to files is the same for all users. Prospero supports user centered naming: users construct customized views of the files that are accessible. A virtual system defines this view and controls the mapping from names to files. Objects may be organized in multiple ways and the same object may appear in different virtual systems, or even with multiple names in the same virtual system.

Prospero directories can contain references to files and directories that are stored on remote nodes. This allows distribution at a much finer level of granularity than is possible in traditional distributed file systems. Prospero also provides several tools to support customization. Among them are the union link and the filter.

A.1 The Directory Mechanism

In Prospero, the global file system consists of a collection of *virtual file systems.* Virtual file systems usually start as a copy of a prototype. The root contains links to files or directories[1] selected by the user. Directory links can be of several types: conventional, union, and filtered.

A conventional link is similar to a hard link in traditional file systems. It may be made to any type of object, including a directory. It maps a name for an object to the information needed to access the object. As long as an unexpired link to an object exists the object may be accessed by that name. If the object moves, a forwarding pointer will allow continued access using the same name. An object is only deleted when no unexpired links to it remain.

A union link can only be made to a directory. With a union link, the objects included in the linked directory become part of the virtual directory containing the link. Thus, the contents of a virtual directory are the union of the collection of conventional links it contains, and the contents of all directories included through union links.

Filters may be attached to either type of link. A filter alters the set of links that are seen in directories whose paths pass through the filtered link. A filter can specify which links are to appear and which are to be ignored, it can change the features of individual links, or it can synthesize new links that are not in the original directory. Filters are written in C and are dynamically linked during name resolution. A filtered link contains a reference to the filter and any arguments required by the filter.

A.2 Using the Prospero File System

This section assumes that Prospero has already been installed on your system. Section A.5 describes the installation procedure.

To use Prospero you must first determine the name of the directory that contains the executables.[2] That directory contains a file called vfsetup.source, the contents of which must be read by the shell. This can be accomplished by `source`ing it. For example, if

[1]To distinguish them from directories and links in traditional systems, and because they are often illusory, directories and links in Prospero are sometimes referred to as virtual directories and virtual links.

[2]By default, the installation directory is */usr/pfs/bin*.

the Prospero file system binaries are stored in */usr/pfs/bin* you should either execute the following command or add it to your *.cshrc* file:

```
source /usr/pfs/bin/vfsetup.source
```

Before you can begin using the Prospero file system a virtual system must be created. Section A.2.4 explains how the Prospero site administrator can create a new virtual system. Most Prospero sites support a *guest* virtual system which can be used until your own virtual system is created.

To use a virtual system, you must first execute the **vfsetup** command to initialize your environment. For example, if the name of your virtual system is *guest* you should execute the following command:

```
vfsetup guest
```

A.2.1 File Names

The slash, colon, pound sign, and open and close parenthesis (`/`, `:`, `#`, `(`, and `)`) are special characters in Prospero. The slash separates components of file names and the colon separates information identifying a virtual system from the name of a file within the virtual system. These characters should not appear in any component of a file name unless they have been quoted.

By default, names are resolved relative to the active virtual system. If a colon (`:`) appears in a name, the name is resolved relative to the name space identified preceding the colon. If the character preceding the colon is a pound sign (`#`), then the name preceding the pound sign will be treated as an alias for a previously specified virtual system. For this reason, virtual systems should not have names ending in a pound sign. A double colon (`::`) is the closure operator. When encountered, the name space identified by the CLOSURE attribute (see Appendix C, Section 1.1) of the object named before the double colon is used to resolve the name that follows.

The pound sign (`#`) is also used to resolve name conflicts when the same component of a name is used by more than one object. In this case the pound sign is followed by the magic number of the desired object. For this reason, unless quoted, the pound sign should not be used in a component of a file name if followed by a number (including sign), and if there are no intervening non-numeric characters between the pound sign and the end of the component.

The pound sign (`#`) is additionally used to indicate that a particular named union link is to be followed when resolving a name, or to indicate that a filter is to be applied. In these cases, the pound sign is the first character of a component in a name and it is followed by the name of the union link to be followed or the name of the filter to be applied. For this reason, unless quoted, the pound sign should not be used as the first character in a component of a file name (unless the full name of the component is `#`).

The open and close parenthesis (`(` and `)`) are used to delimit the arguments to a filter specified as part of a file name. The arguments immediately follow the name of the filter (which itself follows a pound sign). If no arguments are required, the null argument list `()` must be included.

Note that in the current release, quoting of special characters is not supported so they should be avoided in the situations described. This is easily accomplished by avoiding the use of the colon and slash altogether, and by avoiding the use of the pound sign at the start of a name, at the end of a name, or where it is immediately followed by a number.

A.2.2 Using Existing Applications

The Prospero file system is presently implemented as a library. A number of applications have been linked with the Prospero library. The relinked versions may be found in the same directory as the other Prospero binaries, and will appear in your search path once the `vfsetup` command has been executed.

Table A.1: Settings for the PFS_DEFAULT environment variable

Value	Meaning
0	Never resolve names within the virtual system
1	Always resolve names within the virtual system
2	Resolve names within the virtual system if they contain a :
3	Resolve names within the virtual system by default, but treat names beginning with an @ or full path names that don't exist in the virtual system as native file names
4	Resolve names within the virtual system by default, but treat names beginning with an @ as native file names

By default, names which are to be resolved using Prospero must contain or be preceded by a colon (`:`). File names that do not contain and are not preceded by a colon are treated as native Unix file names. The default behavior can be modified by setting the PFS_DEFAULT environment variable. This may be done by using the `venable` command. When enabled, names are resolved relative to the active virtual system by default. Names that are to be treated as native Unix file names must be preceded by an atsign (`@`). `vdisable` will return the default to its original state. The meanings of the values for the PFS_DEFAULT environment variable are listed in Table A.1.

A.2.3 Finding Things

The Prospero file system provides tools that make it easier to keep track of and organize information in large systems. When first created, your virtual file system is likely to contain links to directories that organize information in different ways. As the master copy of each of these directories is updated, you will see the changes. You may customize these directories. The changes you make to a customized directory are only seen from within your own virtual system, but changes made to the master copy will also be visible to you. See section A.2.6 for instructions on customizing a directory.

Users are encouraged to organize their own projects and papers in a manner that will allow them to be easily added to the master directory. For example, users should consider creating a virtual directory that contains pointers to copies of each of the papers that they want made available to the outside world. This virtual directory may appear anywhere in the user's virtual system. Once set up, a link may be added to the master author directory. In this manner, others will be able to find this directory. Once added to the master directory, any future changes will be immediately available to other users.

To add a link to the master copy of any of the shared directories, send a message to your site administrator. The address should be *pfs-administrator* on the primary system for the site. If you are using a virtual system stored at the University of Washington, the address would be *pfs-administrator@cs.washington.edu.*

A.2.4 The Commands

This section steps though the process of creating and using a virtual file system. It assumes that the Prospero file system has been installed on the system being used. Later sections explain how to install the Prospero file system at a new site and on individual systems.

Most of the commands that are specific to the Prospero file system take a debug option. The form is *-D#* where # is an optional integer and specifies the level of detail. The higher the integer, the greater the detail. By itself, *-D* sets the debugging level to 1.

Creating a Virtual File System

A virtual file system is created using the `newvs` command. The `newvs` command is for use by the site administrator[3]. To have a virtual system created, send a message to *pfs-administrator* at your site.

```
    newvs [-v#] [-e] [host [name [home [owner [desc_file]]]]]
```

[3]Systems which are running the release as distributed are part of the University of Washington site. The site administrator is *pfs-administrator@cs.washington.edu.* This applies even if your system is running a server.

Table A.2: Verbosity levels for newvs

Value	Meaning
0 (default)	Prompt for required input
1 (-v)	Explain what is required when asking for input
2	List each action taken
3	Stop before each step
4	Stop before each step and explain the action

The `newvs` command is used to create a new virtual system. If called with no arguments, the user is prompted for the system on which the new virtual system is to reside, the name of the virtual system, the home directory, the owner, and the name of the description file.

The name of the virtual system is the default name with which it will appear in the local site's master list of virtual systems. This name is not automatically exported beyond the local site.

`newvs` will create the virtual system, assign a global name to it, and add the selected name in the master list of virtual systems for the local site. It will also copy the links from the *prototype* virtual system to the newly created one. The *-e* option will suppress this copying, and will leave the new virtual system empty. As a final step, `newvs` will optionally write a description file that may be read by `vfsetup`.

The *-v* option is followed by an integer and sets the verbosity level. The meaning of the verbosity levels are presented in Table A.2.

Initialization and Changing Virtual Systems

```
vfsetup [-n host path , [-r,v] name , -f file]
```

The `vfsetup`[4] command sets up the selected virtual system. It adds the appropriate directories to the search path and sets all necessary environment variables. `vfsetup` can be called in several ways. With no arguments, it reads the file *~/.virt-sys* and uses the information found to access the virtual system description. The *-v* option takes the name of the virtual directory containing the system description. The *-n* option takes the name of a host and the physical name of the virtual directory on that host that contains the virtual system description. The *-f* option takes the name of a Unix file that is to be read in place of *~/.virt-sys*.

It is also possible to set up a virtual system by specifying its name. If the *-r* option is specified, the name is taken to be the default name for the virtual system at the local site. If the name is specified without a modifier, the name is looked up in the /VIRTUAL-SYSTEMS directory of the presently active virtual system (the site default is used if no virtual system is presently active). For example, if the name of your virtual system is *guest* you can set up the virtual system using following command.

```
vfsetup guest
```

Creating Directories

```
vmkdir directory
```

`vmkdir` creates a virtual directory with the selected name and adds a link from its parent directory.

Adding and Deleting Links

```
vln [-u, -s, -m] [-e[am], -n host] oldname newname
```

`vln` adds a new link to a directory. *oldname* is an existing name for the object to which the link is to be made. *newname* is the name of the new link. The *-u* option

[4]Because it changes environment variables, this command only has an affect when its output is read by the shell. If *vfsetup.source* has been `source`ed, then `vfsetup` is an alias which will call the *vfsetup* executable in the appropriate manner.

indicates that the new link is to be a union link. The *-s* option is used to specify a symbolic link.

The *-e* and *-n* options require the specification of the name of the host containing the target. The *-n* option indicates that the native information for the target has been specified. *host* is the name of the host on which the target resides and *oldname* is the name of the target on that host. If the *-s* option has also been specified, then *host* is the name of the virtual system to which oldname is relative.

The *-e* (external) option indicates that the object resides on a host that does not run Prospero. *host* is the name of the host on which the object resides, and *oldname* is the name needed to access the object using the selected access method (e.g. if the access method is anonymous FTP, the name is relative to the anonymous FTP directory on the host).

The -e option can optionally specify the access method to be used and information needed by that method. This information is specified by appending it to the *-e* option itself (e.g. -eNFS,/u1). If no access method is specified, the default is AFTP,BINARY, which will work for most files accessible by anonymous FTP[5].

If the standard input to `vln` has been redirected, the input will be searched for a line of the form "Virtual-system-name: vs-name". If found, *oldname* will be relative to the virtual system which has been read from (closed with) the input. If the *-m* option has been specified, the input will be additionally searched for a line for the form "Virtual-file-name: filename". If found, *filename* will be used in place of *oldname*. The reading of the standard input can be suppressed by specifying the *-a* option, in which case the currently active virtual system will be used.

`vrm link`

`vrm` removes the named link from a directory. It is important to note that `vrm` only removes the link. The object will continue to exist if there are any additional links to it.

[5]Contact *info-prospero@cs.washington.edu* if you need more information on this option.

If there are none, then the object will become subject to garbage collection at a future time.

Listing Directories

If the `venable` command (see section A.2.2) has been executed to set the default name resolution mechanisms to the virtual file system, then the `ls` command may be used to list virtual directories. Otherwise, virtual directories may be listed using the `vls` command.

```
vls [-A,-c,-f,-u,-v] [-a[attribute]] [path]
```

`vls` takes the virtual path name for a file or directory. If the path is for a directory, the links within that directory are displayed. If the path is for any other type of object, then the information for the named link is displayed.

By default, `vls` displays for each link the local component of the path name and the information needed to access the object. In most cases, this is a host and a name relative to that host. A '*' preceding the host name indicates that a filter is associated with the link. The *-v* option causes the object type, and the type of each field to be displayed, and it lists the filters associated with the link. It also prevents the truncation of fields that are too long to be cleanly displayed without the *-v* option.

The *-u* option indicates that union links are not to be expanded. By default, union links are expanded, and the results of that expansion displayed. To see which union links are included in a directory, the *-u* option must be specified.

There are cases when a directory might include more than one link with the same name. One way this can happen is if the directory contains union links. By default, only the first link with a particular name is displayed. The *-c* option tells `vls` to display all links, including those with conflicting names. The name of conflicting links will be followed by a "#" and a number that allows them to be uniquely identified.

The *-f* option causes union links which could not be expanded to be displayed. This option is presently set by default.

The *-a* option indicates that the attributes associated with each file are to be displayed. It also forces the verbose option. If only a particular attribute is desired, the attribute can be specified as part of the *-a* option itself (e.g. -aFORWARDING-POINTER). The *-a* option by itself displays all attributes. The *-A* option is similar to the *-a* option, but it only lists the attributes associated with the link itself, not those associated with the object referenced by the link[6].

Moving Around

```
vcd [-u] path
```

The `vcd`[7] command allows one to change the virtual working directory. If no argument is specified, the home directory is assumed. Otherwise, paths starting with a slash (`/`) are treated as relative to the root of the virtual file system, and other paths are treated as relative to the current working directory. ".." specifies the directory above the current working directory along the active path from the root.

The *-u* option allows one to change ones virtual working directory to a directory included through a named union link.

If the `venable` command (see section A.2.2) has been executed to set the default name resolution mechanisms to the virtual file system, then the `cd` command may also be used to change virtual directories.

```
vwd
vwp
```

The `vwd` command prints the name of the current virtual directory relative to the root of the virtual system. The `vwp` command prints the information describing its physical storage location.

[6]When using `vls` to list the results of an archie query, the *-c* and the *-A* options should be specified.

[7]Because it changes environment variables, this command only has an affect when its output is read by the shell. `vcd` is an alias which will call the *vcd* executable in the appropriate manner.

A.2.5 Retrieving Files

```
vget virtual-file [local-file]
```

The commands described so far allow you to move around the virtual file system, but they do not allow you to access the files that it names. If a program has been linked with the Prospero compatibility library, the program can access files directly.

The `vget` command can be used to explicitly retrieve a file that is accessible by anonymous FTP. The *virtual-file* is the name of the file to be retrieved from the Prospero file system. *local-file* is the real name that you want the file to have in your real current working directory. If *local-file* is omitted the last component of the virtual file name will be used.

If the standard input to `vget` has been redirected, the input will be searched for a line of the form "Virtual-system-name: vs-name". If found, *virtual-file* will be relative to the virtual system which has been read from (closed with) the input. If the -*m* option has been specified, the input will be additionally searched for a line for the form "Virtual-file-name: filename". If found, *filename* will be used in place of *virtual-file*. The -*a* option can be used to suppress the searching of the standard input.

A.2.6 Customizing a Directory

When a change is made to a directory, that change is often visible regardless of the path through which the directory is viewed. There are times when it is desirable to make a change that is only visible when the directory is viewed through a particular path, or from a particular virtual system. Such a change creates a customized view of the directory; the change will not affect the view of the directory when reached through other paths, or from other virtual systems.

To create a customized view of a directory, create an empty directory, add a union link from the directory you just created to the target directory, then remove the link to the old directory and replace it with a link to the directory that was just created.

Links that are added to the customized directory will only be visible through the customized directory, but changes to the target directory will also be visible through the customized directory.

Some directories in your virtual system have already been customized. The root of your virtual system is your own. Changes in the root do not appear in the roots of other virtual systems. Each of the links from the root is also a customized directory. For example, if you add a link to the /authors directory, that link will not be visible to others. Directories at the next level, however, are not customized. Thus, if you add a link to the directory /authors/Shakespeare,William, that change will be visible to others unless you first customize that directory.

You can determine whether a directory has been customized by using the `vls -u` command. That command will show the current directory without expanding union links. Admittedly, this is a little confusing. Future releases will support the concept of an owning virtual system. This will clear up some of the confusion by allowing automatic creation of a customized directory when one is needed.

To have a link added to the master copy of a shared directory you should send a message to *pfs-administrator* at your site, or to *pfs-administrator@cs.washington.edu*.

A.2.7 Attributes and Forwarding Pointers

This release includes preliminary support for attributes and forwarding pointers. This means that file attributes may be retrieved using the *-a* option to the `vls` command. User defined attributes may also be associated with a file. For the time being, such attributes must be added manually.

Forwarding pointers are also supported. If a file or directory has moved and a forwarding pointer exists, the forwarding pointer will be returned and the request retried. Like attributes, forwarding pointers must be added by hand. There are not presently any programs to add forwarding pointers or attributes. For information on how to add these by hand, send a message to *info-prospero@cs.washington.edu*.

Table A.3: Protection modes in access control lists

Directory		Link	
Character	Meaning	Character	Meaning
A	Administer	a	Administer
V	View	v	View
L	List	l	List
R	Read	r	Read
M	Modify	m	Modify
D	Delete	d	Delete
I	Insert	does not apply to link	
B	Administer	does not override link	
Y	View	does not override link	
>	Add-rights	]	Add-rights
<	Remove-rights	[	Remove-rights
)	Add-rights	does not override link	
(	Remove-rights	does not override link	

A.3 Protection

Access control lists (ACLs) may be associated with directories in Prospero, and with individual links within a directory. These access control lists specify how directory information is to be protected. They have nothing to do with the protection of the file to which a link refers. Table A.3 lists the protection modes that may appear within an access control list entry.

When an entry appears in both columns, it means that the entry on the directory overrides the entry on a link. For example, a user with R access to the directory can read a link even if denied r access to the link. If a link permission is stored in a directory access control list, then that permission indicates the default protection associated with links in the directory that do not specify their own access control lists.

The administer permission allows the changing of the access control list. View allows it to be viewed. List allows one to see a directory entry using wildcard searches. Read is required to determine the binding of a link (the file it references). If one is allowed read, but not list, then one can only retrieve the link if its exact name is specified. Modify allows one to change the binding of a link, but it does not allow one to add or delete links. Insert allows links to be added, and delete allows them to be deleted. The add and remove rights are a restricted form of administer. They allow an individual to add or remove a restricted set of rights. For example, ">r" allows one to grant read access to someone else without also allowing one to grant modify access.

Negative rights may be specified by prepending a minus sign (-) to the rights field. The order of access control list entries is important. For negative rights to have an effect, they must precede any rights that authorize access by that individual. When new ACL entries are added, they are added at the front of the list, meaning that recent entries take precedence over older ones.

When access is checked for a link, three access control lists are checked. First, the ACL associated with the link itself is checked. If the link does not have its own ACL, then the default ACL associated with the directory is used. Next, the ACL associated with the directory is checked to see if it grants rights that override those in the link. Finally, if access has still not been granted, a special override ACL is checked. This is maintained by the system and should be used only in emergencies. Negative rights in one list does not override access granted in another.

There are several different ACL entry types. DEFAULT, SYSTEM, and DIRECTORY cause other access control lists to be included. DEFAULT is the default ACL specified by the system on which the Prospero server is running. SYSTEM is also specified separately on each Prospero server. It usually grants additional access to system administrators. DIRECTORY is the default access control list associated with the directory containing a link. It allows one to easily specify that the rights on a link are to be in addition to the default rights specified in the directory. If no rights are associated with these ACL entry types, then the rights granted are based on those in the DEFAULT, SYSTEM, or DIRECTORY access control lists themselves. If rights are specified for such entries, then the rights are the minimum of those specified, and those in the included ACL. The ACLs for new files and directories allow access to DEFAULT and SYSTEM as defined in those lists. Users have the option of removing such access[8].

The ANY, AUTHENT, ASRTHOST, and TRSTHOST ACL types grant rights to the specified individuals according to the accompanying permission list. ANY matches any user. AUTHENT specifies an authentication method, and the name of the authorized individual as returned by that method. At present, this entry type is not supported. The ASRTHOST method specifies a list of authorized principals in the form *user@internet-address*. If no Internet address is specified (and no atsign), then the user is matched regardless of the requesting host. Octets of the Internet address can be wildcarded, or replaced with a '%'. A wildcard matches any number, and a '%' matches the number corresponding to the local host. For example, "bcn@%.%.%.*" matches the user "bcn" on the local subnet.

The ASRTHOST type accepts the username asserted by the client. It is not possible to verify that the user has not modified the software to claim someone else's identity. The Internet address can generally be considered accurate, though it too can be spoofed by a knowledgeable and determined attacker. The TRSTHOST type is identical to the ASRTHOST type, but is accepted only when the request originates from a privileged port on the requesting system. Although this method might be used to provide security similar to that for the Berkeley R commands, it is not recommended that you install Prospero binaries setuid root until the sources have undergone careful scrutiny for possible security holes[9].

Listing ACLs

The `list_acl` command may be used to list the contents of an access control list.

```
    list_acl [-d dir] [link-name]
```

[8]The SYSTEM access control list is separate from the OVERRIDE list which can not be removed.

[9]By no means should vget or vcache be installed setuid root as these command write files to paths specified by the user.

With no arguments, it lists the ACL for the current directory. If the *-d* option is specified, the argument that follows the option specifies the directory whose ACL is to be listed. An optional link-name specifies that the ACL to be listed is that of the named link within the directory.

Modifying ACLs

The `set_acl` command allows one to change the access control list associated with a link or directory.

```
set_acl [-asirKE,-n,-N] [-t type] [-d dir] [-l link] rights principals
```

The *-d* option is followed by the name of a directory. By default, the ACL for the directory is modified. The *-l* option allows one the modify the ACL for an individual link.

The *-a, -s, -i, -r, -K,* and *-E* options indicate the operation to be performed on the ACL. They correspond in order to add rights, subtract rights, insert a new entry, remove an entry, kill the entire ACL (setting it to the default), and replacing the entire ACL.

If the *-t* option is specified, it must be followed by the type of the ACL entry to be added, deleted, etc. If the *-t* option is not specified, ASRTHOST is the default.

The first field following the options specifies the rights to be added or deleted. All remaining arguments are the names of the principals to be included in the particular ACL entry.

When an ACL is set to an initial value (using the *-K* or *-E* options), the SYSTEM ACL is automatically included. The SYSTEM entry can be removed by using the *-r* option in a subsequent `set_acl` command. The addition of the SYSTEM entry can be suppressed by using the *-n* option in conjunction with the original *-K* or *-E* option. Whenever rights are removed from an ACL, the system checks to make sure that the user removing the rights will be able to fix any mistakes. If the the change would result in the user being unable to make subsequent changes, the minimal rights allowing the user to make subsequent changes are automatically added back. This safety mechanism may be overridden by specifying the -*N* option.

A.4 Filters

Filters are written in C, compiled, and dynamically linked during name resolution. Because of portability problems with the dynamic linker, filters are not included in this release, but are available upon request.

A.5 Setting up a new system

This section explains how to install the Prospero file system on a new system. It assumes that the local site has already been configured, This section (Section A.5) can be skipped by most users.

A.5.1 Building the Binaries

[See the INSTALLATION file in the distribution]

A.5.2 Running the Server on Unix (like) Systems

If you are installing the Prospero server, a user and group ID must be established under which the directory server will run. The directory associated with the user ID should be a location in which additional information about virtual files and directories can be stored. New files which are to exist only within the Prospero file system will also be stored under this directory. It is suggested that you chose the user name *pfs* for this pseudo-user, but other names may be used as well.

Once the user ID has been set up, install the binaries. The directory in which they must be installed is selected at compile time. As originally distributed, it is */usr/pfs/bin*. The program `pstart` should be installed setuid and setgid the pseudo-user just described.

```
pstart [hostname]
```

To start the server run `pstart`. `pstart` takes an optional host name. If specified, the host name must be the primary name for the host on which the server is running. In most cases, the server is able to determine the name on its own and there is no need to specify it as an argument.

`pstart` will connect to the directory associated with the pseudo-user, it will check to make sure that the user id is set appropriately, and it will exec the directory server with the appropriate arguments.

Although it is not recommended, the directory server can also be started manually. You must first be logged in as (or be su'ed to) the user under whose ID you want the server to run. You can then execute `dirsrv` passing as arguments the required directory names.

```
dirsrv [-m] root shadow data aftpdir afsdir hostname
```

The *-m* (manual) option prevents the directory server from dissociating itself from the terminal. It is only useful for debugging. *root* is the logical root of the system. Only files below this point (and those under aftpdir and afsdir) will be accessible through the Prospero file system. *shadow* is the name of the directory that is to contain additional information about files and directories. It should typically be the *shadow* subdirectory of the pseudo-user described above. *data* is the local directory under in which new virtual directories and their contents will be stored.

aftpdir is the name of the directory hierarchy to which anonymous FTP has access and *afsdir* is the name of the directory through which files from the Andrew File System may be accessed. If these access methods are not supported by your system, these arguments should be the null string.

Users can use the Prospero file system even if the server is not running, but they will be unable to access files or directories stored locally. If `pstart` is installed setuid and setgid to the Prospero user and group IDs, then the directory server can be started by any user. You may also want to start the directory server from the system's */etc/rc* file.

As things stand, users can access files and directories created on the local system, but they can not create new virtual systems stored locally. If you want to allow virtual systems to be stored locally, then you must have the site administrator add a reference to the new system from the *pfs_storage* virtual directory.

Adding References to the New System

The remainder of this section describes the actions that are to be taken by the site administrator. Systems that are running the release as distributed are part of the University of Washington site. The site administrator is *pfs-administrator@cs.washington.edu.* This applies even if your system is running a server. If you are not a site administrator, you can skip this section.

If you want to allow virtual systems to be stored locally, several links must be added and a new virtual directory must be created. This will only be possible if the server has **not** been configured read-only. When a new site is established (see Section A.6) these links and directories are automatically created on the primary system for the site. The following steps are only required when adding additional systems.

In the following steps, HOST is the fully qualified domain name for the host to be added and PATH is the full path of the subdirectory of the pseudo-user (described above) which will store the new virtual directories. The last component will typically be *pfsdat.* If that directory does not already exist, it should be physically created.

A link must be added from the virtual directory *pfs_storage* to the pfsdat directory on the new site. While still in the master virtual system, the following steps will add this link.

```
vcd /pfs_storage
vln -n HOST PATH HOST
```

You will next have to create the *local_vsystems* virtual directory. You do this by issuing the commands:

```
vcd HOST
vmkdir local_vsystems
```

A.6 Setting up a New Site

This section explains how to set up a new site. Systems that are running the release as distributed are part of the University of Washington site. This applies even if your system is running a server. If you would like to set up your own site, send a message to *info-prospero@cs.washington.edu* to obtain the appropriate distribution (several additional files). Unless you are setting up your own site, you may skip the remainder of this section.

Before you begin, you will have to obtain a global prefix that will uniquely identify the virtual systems registered at your site. A prefix may be obtained by sending a message to *pfs-administrator@cs.washington.edu.*

The global prefix is part of the low level name for each virtual system. It should be thought of as an address. Users employ higher level names to specify virtual systems. Registering a prefix will allow objects created at your site to be named by others. Even if your site will not be reachable by any other sites, it is still important to register a prefix. Doing so guarantees that no other site has the same prefix. This will make it possible to connect with the rest of the global system should a connection ever be established. More information on setting up an isolated site is described in Section A.6.2.

A.6.1 Setting up the Master Directories

To initially set up a new site, run the command `newpsite`. `newpsite` will construct a skeletal site configuration based on the compile time options described in the previous section.

Once the site has been set up, it will be necessary to create additional directories and add them to the prototype on which new virtual systems will be modeled.

A.6.2 Setting up an Isolated Site

As was already mentioned, even if your site will be isolated from others, you should still try to register a unique global prefix.

If your site will be isolated from other sites you will have to set up a replica of the global root. This directory must be reachable with the name "#" from your site's master list of virtual systems. To create a replica of the global root, create a directory and add nested subdirectories corresponding to each component of your sites global prefix. The last entry should be a link to your site's master list of virtual systems.

If You Cannot Register a Prefix

If it is not possible to contact *pfs-administrator@cs.washington.edu,* then it may still be possible to generate a unique global prefix based on a unique identifier assigned by another authority. Right now, the only names that may be turned into unique global prefixes are officially registered Internet domain names.

Internet Domain Names. If you have an officially registered Internet domain name, it may be turned into a global prefix by reversing the order of the components, replacing the periods with slashes, and prepending "#/INET/".

```
ISI.EDU      =      #/INET/EDU/ISI
```

A.7 Glossary

conventional link. A conventional link is similar to a hard link in the Unix file system. It maps a name for an object to the information needed to access it.

filter. A filter is a program attached to a link. A filter can modify the results of directory queries where the path from the root of the virtual file system to the queried directory passes through the filtered link.

global file system. The global file system is the collection of links and directories that make up the virtual file systems accessible to the user. The links and directories form a generalized directed-graph.

link. A link is either a conventional link or a union link, with or without an attached filter.

local system. The local system is the physical system to which a user is logged in, on which processes execute, or on which files are stored.

master directory. A master directory is a directory maintained at the site level, and included through union links as part of the corresponding directory in multiple virtual systems.

site. A site is a collection of virtual systems administered by a particular organization. An important characteristic of a site is that its virtual systems contain prominent references to the other virtual systems that are part of the site.

union link. A union link is a link to a directory that causes the links that are part of the linked directory to appear as part of the directory containing the union link. A directory's contents are the union of the set of conventional links it contains and the contents of all directories included through union links.

view. A view is a mapping from names to objects. Name spaces, parts of name spaces, and individual directories all define views. Because it specifies a name space, a virtual system also imposes a view. It is possible for more than one virtual system to impose the same or similar views.

virtual directory. A virtual directory is a directory in a virtual file system. The contents of a virtual directory might be calculated at the time the directory is queried by applying filters or expanding union links.

virtual file system. A virtual file system is the file system part of a virtual system. It consists of a root directory and all the files and directories that can be reached by traversing 0 or more links. The virtual file system is a projection of the global file system as viewed from the selected root.

virtual system. A virtual system is a distributed system that is assembled from the files, processors, services, applications, users and other components available over a global network. The owner of a virtual system identifies the components of interest, and assembles them into a virtual system by assigning names.

A.8 Quick Reference

Commands affecting the Prospero file system:

```
vfsetup [-n host path , [-r,v] name , -f file]

vcd [-u] path

vwd

vls [-v] [-u] [-f] [path]

vln [-u] [-s] [-e] [-n host1] name1 name2

vmkdir directory

vrm link

vget virtual-file [local-file]

newvs [-v#] [-e] [host [name [home [desc_file]]]]

list_acl [-d dir] [link-name]

set_acl [-asirKE,-n,-N] [-t type] [-d dir] [-l link] rights principals

vdisable

venable
```

Meanings for PFS_DEFAULT:

The meanings of the values for the PFS_DEFAULT environment variable are:

Table A.4: Settings for the PFS_DEFAULT environment variable

Value	Meaning
0	Never resolve names within the virtual system
1	Always resolve names within the virtual system
2	Resolve names within the virtual system if they contain a :
3	Resolve names within the virtual system by default, but treat names beginning with an @ or full path names that don't exist in the virtual system as native file names
4	Resolve names within the virtual system by default, but treat names beginning with an @ as native file names

Appendix B

The Prospero Library

This appendix describes the entry points to the Prospero library.

B.1 Pcompat Library

The compatability library includes replacements for existing system calls and library routines that interact with the directory service. The replacements optionally resolve names using the Prospero file system. The behavior depends on the value of the `pfs_enable` global variable. Possible values are defined in *pcompat.h* and are described below.

Table B.1: Settings for the `pfs_enable` global variable

Value	Meaning
PMAP_DISABLE	Never resolve names within the virtual system
PMAP_ENABLE	Always resolve names within the virtual system
PMAP_COLON	Resolve names within the virtual system if they contain a :
PMAP_ATSIGN_NF	Resolve names within the virtual system by default, but treat names beginning with an @ or full path names that don't exist in the virtual system as native file names
PMAP_ATSIGN	Resolve names within the virtual system by default, but treat names beginning with an @ as native file names

Entry points: closedir creat, execve, open, opendir, readdir, scandir, seekdir, stat, telldir, pfs_open, pfs_fopen, and pfs_access.

closedir, **creat**, **execve**, **open**, **opendir**, **readdir**, **scandir**, **seekdir**, **stat**, and **telldir** are identical to the entry points with the same names in the standard C library except that, depending on the value of the `pfs_enable` variable, file names may be resolved using Prospero.

pfs_access(*path,npath,flags*) accepts a name, *path*, that is to be resolved using Prospero. **pfs_access** resolves the name, selects an access method, mounts the appropriate file system or retrieves the file if necessary, and returns a new name in *npath* that may be passed to open. *npath* must be a buffer large enough to hold the new name. By setting *flags*, it is possible to specify that the file is to be created if it does not exist (PFA_CREATE), or to indicate that the file will be opened read only (PFA_RO). **pfs_access** returns PSUCCESS (0) or PMC_DELETE_ON_CLOSE on success. A return value of PMC_DELETE_ON_CLOSE indicates that the file has been cached on the local system and that the calling application should delete the cached copy when done with it. Any other return code indicates failure.

pfs_open(*vl,flags,mode*) and **FILE *pfs_fopen**(*vl,type*) are identical to **open** and **fopen** in the C library except that instead of a filename, they take a pointer to a Prospero virtual link structure and open the file referenced by the link.

B.2 PFS Library

The PFS library includes procedures for allocating and freeing Prospero data structures, resolving names using Prospero, reading directories, retrieving attributes, adding and deleting links, and creating directories. Only those procedures needed by application programmers are described here. The remaining routines are called internally.

Entry points: atalloc, atfree, atlfree, vlalloc, vlfree, vllfree, add_vlink, del_vlink, get_vdir, mk_vdir, pget_am, pget_at, rd_vdir, and rd_vlink.

atalloc() and **vlalloc**() allocate structures for storing attributes and virtual links, returning NULL on failure. **atfree**(*at*) and **vlfree**(*vl*) free the storage allocated to *at* and *vl*. **atlfree**(*at*) and **vllfree**(*vl*) free *at* and *vl* and all successors in a linked list of such structures.

add_vlink(*direct,lname,l,flags*) adds a new link *l* to the directory named *direct* with the new link name *lname*. *direct* is a string naming the directory that is to receive the link. If *flags* is AVL_UNION, then the link is added as a union link. **add_vlink** returns PSUCCESS (0) on success and an error code on failure.

del_vlink(*path,flags*) deletes the link named by *path*. At present, *flags* is unused. **del_vlink** returns PSUCCESS (0) on success and an error code on failure.

get_vdir(*dhost,dfile,components,dir,flags,filters,acomp*) contacts the Prospero server on host *dhost* to read the directory *dfile*, resolving any union links that are returned and applying *filters*. If *components* is a non-null string, only those links with matching names are returned. *dir* is a Prospero directory structure that is filled in. *flags* can suppress the expansion of union links (GVD_UNION), force their expansion (GVD_EXPAND), request the return of link attributes (GVD_ATTRIB), and suppress sorting of the directory links (GVD_NOSORT). *acomp* should normally be NULL. One should normally not need to call this procedure. Use **rd_vdir** and **rd_vlink** instead. The arguments to this procedure will be changing in a future release. **get_vdir** returns PSUCCESS (0) on success and an error code on failure.

mk_vdir(*path,flags*) creates a new virtual directory with the new name *path* in the currently active virtual system. *flags* is not presently used. **mk_vdir** returns PSUCCESS (0) on success and an error code on failure.

pget_am(*link,ainfo,methods*) returns the access method that should be used to access the object referenced by *link*. *ainfo* is a buffer that is filled in with additional information required for the selected access method. *methods* is a bit-vector identifying the methods that are acceptable to the application. The methods presently supported are anonymous FTP (P_AM_AFTP), Sun's Network File System (P_AM_NFS), and the Andrew File System (P_AM_AFS). **pget_am** returns P_AM_ERROR (0) on failure and leaves an error code in `perrno`.

PATTRIB pget_at(*link,atname*) returns a list of values of the *atname* attribute for the object referenced by *link*. If *atname* is NULL, all attributes for the referenced object are returned. **pget_at** returns NULL on failure, or when no attributes are found. On failure, an error code is left in `perrno`.

rd_vdir(*dirarg,comparg,dir,flags*) lists the directory named by *dirarg* (relative to the current working directory or the root of the active virtual system) returning the links whose names match *comparg*. *dir* is a Prospero directory structure that is filled in. *flags* can suppress the expansion of union links (RVD_UNION), force their expansion (RVD_EXPAND), request the return of link attributes (RVD_ATTRIB), suppress sorting of the directory links (RVD_NOSORT), suppress use of cached data when resolving names (RVD_NOCACHE), or request the return of a reference to the named directory, suppressing the return of its contents (RVD_DFILE_ONLY). **rd_vdir** returns PSUCCESS (0) on success and an error code on failure.

VLINK rd_vlink(*path*) is an alternative interface for resolving names. **rd_vlink** returns the single link named by *path*. Its function is equivalent to calling **rd_vdir** with comparg set to the last component of the path and dirarg set to the prefix. **rd_vlink** returns NULL on failure leaving an error code in `perrno`.

Appendix C

Directory and File Attributes

This appendix describes the object and directory information maintained by the Prospero file system.

C.1 Objects

A Prospero object is any item that can be referenced in a directory. The most common kinds of objects are files and directories. An object has a native representation and a set of attributes describing it.

C.1.1 Attributes

Prospero maintains the following attributes about the objects it maintains.

- **Closure.** The name of the virtual system to be used when resolving names embedded in the object.
- **Owning virtual system** (optional). Used to decide whether changes should be customizations or modifications.
- **Owner.** The principal responsible for the object.
- **Access control information** (optional).

- **Last writer, write date, etc.**
- **Version info** (optional). Number of versions to keep, version number, etc.
- **Attributes or keywords** (optional). User specified.
- **Short description** (optional).
- **Locks** (optional).
- **List of replicas** (optional).
- **Replication type** (optional).
- **Other replication information** (optional).
- **Access methods**.
- **Time to live**. The lifetime for newly created or refreshed links to the object.
- **Time in future when TTL expires.** This is the TTL plus the time that a link to the object was last created or refreshed.
- **Back links**. A possibly incomplete list of directories with links to the object.

Note that the object's name is not one of its attributes. The object's name is the concatenation of the names starting from the active virtual system. An object may have different names in different virtual systems, or even multiple names within a single virtual system[1].

C.1.2 Persistence

An object continues to exist until the last non-expired link referencing the object is removed. If a user wishes to recover the storage space for an object, it is flagged for

[1]Despite this, well known names from agreed upon starting points might be entered as user-defined attributes for objects.

removal. When links to the object are refreshed, notification is sent that the object is about to disappear, and anyone wanting to maintain their reference must copy the object elsewhere. Subsequent users have the option of making their own copy, or updating their link to refer to one of the new copies.

C.1.3 Mobility

Objects may move from one site to another. If they do, a forwarding pointer must remain at the old location until the time-to-live expires. This enables anyone with a non-expired link to the object to refresh the link and record the object's new location.

C.2 Directories

A directory is an object and as such, everything that applies to objects also applies to directories. The physical representation of a directory is interpreted by the Prospero server on the system storing the directory. The Prospero protocol defines the interface through which a directory is accessed.

C.2.1 Additional Attributes

The Prospero implementation maintains the following additional attributes about the directories it maintains.

- **Directory format version number**.
- **Access control information**. This includes an access control list for the directory itself and a default access control lists for links within the directory.
- **Links to the objects included in the directory**.
- **Anything else required by the local directory server**.

C.2.2 Link Attributes

Prospero directories contain links to the objects that are included in the directory. The following information is maintained for each link.

- **Name.** This is the single component of the object's path relative to the current directory.

- **Short description of link** (Optional).

- **Type** (*union/normal*). The type of a link can be either normal [L] or union [U]. In the case of a normal link, an entry for the link is visible when the directory is listed. In the case of a union link, which can only be made to another directory, the links from the target directory appear as part of the directory from which the union link originates when the originating directory is listed. If multiple objects have the same name, the order in which the union links appear determines which object is visible.

  Two types of links which may appear locally (either in the server, or on the client) are [-] deleted, and [N] native. It is not legal for these codes to be sent across the network. Type [N] links are converted to [L] and type [-] are skipped entirely.

  Two additional types of links are reserved for future use. They are replicate [R] and propagate [P]. A replicated link is a link to a replica of the present directory. All changes should be propagated to the replica. A propagate link is like a replica except that only some of the links are replicated. Changes should be propagated, but received changes from other directories should only be used to update the information on existing links, not to create new ones. Directory replication is approximate, and it may take a while for changes to propagate. If consistency is required for certain links, then P links instead of R links should be used, and the important links should only appear in one place. In the future, an alternate mechanism will be available to provide strong consistency for replicated directories.

- **Hidden/Not-Hidden/Externally-Hidden** By default, a hidden link is not displayed when a directory is listed. It is, however, returned by the directory server, and is traversed if the actual name is specified in a pathname. An Externally-Hidden link is a hidden link that will be displayed if the current virtual system is the same as the owning virtual system for the directory containing the link. The user may override the hidden option, causing hidden links to be listed. Note that it is also possible to hide a link by specifying its protection as non-listable. Such links will **only** be returned by the directory server when the actual name of the link has been specified.

- **Filter** (*program, data*) (optional). Multiple filters allowed. The filter attribute is a pointer to a program that can be used to filter the contents of directories to which links are made. *data* is the set of optional arguments passed to the filter program. In addition to the linked directory and arguments, the filter has access to all other information available from the current directory.

  Filters come in several types. By default, a filter is a directory filter, and is applied when searching directories. A hierarchy filter is similar to a directory filter. The difference is that a directory filter is applied to a single directory, while a hierarchy filter is applied to the entire hierarchy (including subdirectories) reached through a link. Directory and hierarchy filters come in two types. The default is post-expansion. The filter is applied after all union links have been expanded. A pre-expansion filter is applied before union links are expanded. An object filter is applied when accessing an object other than a directory, and might be used, for example, to cause some operation to be performed on the object before it is accessed. Note, however, that all types of filters are associated with links.

  Filters are applied to a directory in the order of pre-or post expansion, decreasing depth of the links to which they are attached (for hierarchy filters), and finally in the order that the filters are specified on the traversed links.

- **Attributes** (optional). User defined attributes to be associated with the link. This allows a user to add (or override) attributes to the linked object (which might be owned by someone else, and thus not modifiable).
- **Destination host name.** The name of the host on which the file can be found.
- **Destination host type.** The type of the hostname. In particular, whether the hostname is an Internet address, a domain style name, or a name in some other naming system.

  Note that a symbolic link is a link where the destination host is a virtual system, and the destination object name is a name relative to that virtual system.
- **Destination object name.** The name of the object relative to the destination system.
- **Destination name type.** The type of destination name. Different types of names include file IDs, names relative to the root of the local file system, etc.
- **Version number** (optional). By specifying a version number in a directory link, the link is made to a specific version of the object, and changes to the object will not be visible through the link.
- **Access control info** (optional). Allows restrictions on who can read or modify individual directory entries.
- **Destination expiration date.** This entry indicates how long the information in the link should be considered valid. When an object is accessed through a link, the destination expiration date should be set to the current time plus the destination time-to-live.

  Note that expired directory entries do not disappear. Typically they remain valid. The expiration means that there is no longer a guarantee that the object originally referenced can still be found.

- **Object identifier**. When a new object is created, a unique object identifier may be assigned. This identifier can be included in links, and used to further verify that the named object is the one that is actually desired. It allows one to reference objects after their expiration dates with the guarantee that if these identifiers match, it is the same object. In the prototype, the object identifier is a random number.

- **Valid-till** (optional). If a link is a cached value, then this field indicates how long the entry should be considered valid. For example, a symbolic link may have a corresponding cached hard link. Until its expiration a cache entry may be used instead of searching based on the symbolic link. If this field is 0, then the link is not a cached entry.

- **Last-update** (optional). This is the time the link was last updated. Its expected use is for resolving conflicting updates in replicated directories.

C.2.3 Replication

When support for replication is added to Prospero, a directory might include multiple links with the same name for the same object. Not all replicas need be listed, however, because each replica will maintain its own list of siblings. Multiple entries are important to increase reliability and availability.

Appendix D

The Prospero Protocol

This appendix describes the Prospero protocol. Communication with directory servers uses a reliable datagram protocol described in Appendix E. Requests and responses are human readable commands and multiple commands may be sent in a single message.

Prospero is implemented on top of a datagram protocol to reduce the overhead that would otherwise be incurred when establishing connections to multiple directory servers. The decision to use a datagram protocol directly, instead of through a higher level mechanism such as remote procedure call, was made for reasons of portability. I did not want Prospero to depend on another protocol, software package, or other resource, unless that resource was almost universally supported by every computer on the Internet.

The use of humanly readable commands in the protocol has several advantages. It eliminates problems with byte ordering, and it makes future changes or additions to the protocol easier to incorporate while maintaining compatibility across versions; new commands or additional options to existing commands can be easily added.

The ability to request multiple operations in a single message improves performance, and it provides a simple way to request that a collection of operations (on a single server) be performed atomically.

D.1 Commands

Commands are separated by the <LF>. Items sent in response are separated by <LF>s, commas, spaces, or tabs. Characters (or null fields) can be quoted using single quotes ('). While inside quotes, a quote can be included by doubling it. If multiple options are to be specified for a single command, the options are separated by a "+".

D.1.1 VERSION <version-number> <software-id>

This command requests or specifies the protocol version number. If the specified protocol version number is not supported, the response will be:

```
VERSION-NOT-SUPPORTED TRY xx-yy,zz
```

Where xx-yy and zz lists those versions that are supported. If no arguments are supplied (or if the argument is not a number), then the VERSION command will return the current protocol version number being used by the server in the form.

```
VERSION <number>
```

The software identifier is optional, but if specified, it identifies the software and version of the software that is generating the request.

D.1.2 AUTHENTICATOR <type> <authenticator>

This command authenticates the principal making the request. The type is the type of the authenticator. It might be password, a Kerberos authenticator, or data used by an alternative authentication mechanism.

D.1.3 DIRECTORY <id-type> <object-id> [<version> [<magic-no>]]

This command specifies the directory to be read or modified by the commands that follow as part part of the same request. This command does not generate a response unless the selected directory is not a directory. In that case, the response:

```
NOT-A-DIRECTORY (being phased out) or
FAILURE NOT-A-DIRECTORY
```

is returned, and the remainder of the request ignored.

D.1.4 ATOMIC

This command specifies that all commands that follow are to be completed atomically. If one of the commands fails, then none of the commands are to have an affect. This may require undoing the affects of commands which have already completed. Note that this command works only for requests issued in the same message, and all commands must be executable on the single server to which the message was sent.

D.1.5 LIST <options> COMPONENTS <optional-component>

LIST is used to look up information stored in a directory. <optional-component> is the name of the component relative to the directory specified in the DIRECTORY command. The optional component can include wildcards, or can include multiple names separated by a space. If a space is to be included in a name, the name must be quoted.

Multiple component names are allowed. The multiple components within a single name are separated by the slash (`/`). If the result of the lookup of one component is another directory at the same storage site, then the directory server will repeat the process and look up the next component. When a lookup results in a dead-end, or a directory at another site, the server will return an entry indicating which components remain to be resolved by the client.

<options> contains a list of options to the LIST command. Among the options supported are EXPAND which tells the remote directory server to expand any union entries in the directory. By default, the response will contain the names of union links to be included, and the client must submit a new query. A directory server can ignore the EXPAND option to the list command if it so desires. The LEXPAND option is the same as EXPAND, but indicates that the server should only expand union entries for local

directories. The LEXPAND option is implied if a multiple component name is being resolved.

The VERIFY option tells the remote server that the purpose of the query is to verify whether the remote directory exists and is a directory. No components are to be returned.

The ONECOMP option tells the remote server that the component name should be treated as a single component. This option is required only when a component of a name includes a slash.

The ATTRIBUTES option indicates that the attributes associated with the link are to be returned. ATTRIBUTES must be the last specified option and is followed by the names of the attributes to be returned as additional options (i.e. they are separated by the +). If no attributes are listed, then all attributes are returned. If this option is specified, but the object resides on a different host, then the attributes stored with the link are returned, not those stored with the object referenced by the link.

On failure, a standard error responses will be returned. On success, the response will be a sequence of entries containing information about the requested files.

```
LINK <L/U/R/P> <type> <component> <host-type> <host-name> <id-type>
     <object-id> <version> <magic-no> (opt: DEST-EXP <dest-expiration>
     LINK-EXP <link-expiration>)
```

An UNRESOLVED response lists the components of the name that must be further resolved relative to the returned link.

```
UNRESOLVED <components>
```

If a filter is returned, it applies to the most recent link or union link that was returned.

```
FILTER <filter-type> <filter-host-type> <filter-host-name>
       <filter-id-type> <filter-object-id> <version> <magic-no>
       (opt: ARGS <filter-args>)
```

The types of filters are: D=DIRECTORY, filter is applied to the current directory; H=HIERARCHY, filter is applied to all directories reachable through the filtered link; O=OBJECT filter is applied to an object other than a directory; U=UPDATE, filter is applied when updating the directory. If union links are to be expanded, filters are normally applied after the expansion. A lower case type code (d, or h) indicates that the filter is to be applied before any union links are expanded.

The optional list of arguments to a filter is a single string. If there are multiple arguments, then they are separated within the string by spaces. If there are multiple arguments or if args contain special characters, then the list of arguments must be quoted so that it will be passed as a single string.

If attributes were requested, the link will be followed by lines that specify the values of the requested attributes. The form of the response will depend on whether the attribute is associated with the link, or with the actual object. If the attribute is associated with the link, the applies-to field indicates whether it applies to the LINK or the object, and if the object whether it is a CACHED attribute, a REPLACEMENT attribute, or an ADDITIONAL attribute. In case of conflicts between attributes associated with the link and those associated with the object, a cached attribute is superseded, a replacement attribute takes precedence, and an additional attributes leaves both intact.

```
OBJECT-INFO <attribute> <type> <value> <optional-info>
LINK-INFO <applies-to> <attribute> <type> <value> <optional-info>
```

If the value of an attribute is a link, the link might itself have attributes. When returned the names of such sub-attributes are enclosed in parenthesis to the appropriate nesting level. If the name of an attribute is to begin with an open parenthesis, the opening parenthesis must be quoted.

The order in which attributes are returned is significant. Some attributes might span multiple lines in which case, each line might appear as a separate attribute. This does not preclude the inclusion of a multi-line value in a single attribute if the value is appropriately quoted, but the existing implementation does not presently support such a representation.

If no files are found, the reply will be:

```
NONE-FOUND
```

D.1.6 LIST-ACL <options> <optional-component>

This request is used to list the access control list for the directory specified in the previous DIRECTORY command, or for a link within the directory. The optional component is required only if requesting the ACL for a link within the directory (option = LINK). It is to be left out when requesting the ACL for the directory itself (option=DIRECTORY). The response will be zero or more lines for the form:

```
ACL <entry-type> <authentication-type> <rights> <principals>
```

D.1.7 GET-OBJECT-INFO <requested-attributes> ID <id-type> <file-id> <version> <magic-no>

This command requests information about an object. <requested-attributes> is a list of those attributes that are desired. Valid attributes include ACCESS-METHODS, CLOSURE, DESCRIPTION, FORWARDING-POINTER, KEYWORDS, LAST-REFERENCED, LAST-WRITER, LOCKS, OWNER, REPLICAS, SIZE, STORAGE-LOCATION, TTL, TTL-EXPIRES. VERSION-NUMBER, VIRT-SYS, WELL-KNOWN-NAMES, and WRITE-DATE. ALL indicates that all attributes are to be returned. Multiple attributes may be separated by commas (with no intervening white space). The above are reserved attributes. User defined attributes are also allowed and are accessed the same way.

The response can be multiple lines, each containing a value for an attribute. The form of the response will be.

```
OBJECT-INFO <attribute> <type> <value> <optional-information>
```

The order in which attributes are returned is significant. Some attributes might span multiple lines, in which case each line might appear as a separate attribute. This does not preclude the inclusion of a multi-line value in a single attribute if the value is appropriately quoted, but none of the existing implementations can deal with such a representation.

D.1.8 EDIT-OBJECT-INFO <id-type> <object-id> <version> <magic-no> ATTRIBUTE <attribute> <optional-type-modification> <type> <value>

This command is used to change information about a file. <attribute> is the attribute about the file that is to be changed. Some attributes involve adding a new instance, deleting an instance, or replacing an instance. In these cases, the <optional-type-modification> will be either ADD, DELETE, or REPLACE. The new value is <value>.

D.1.9 CREATE-LINK <L/U> <component> <link-type> <host-type> <host-name> <id-type> <object-id> <version> <magic-no>

This command creates a new link in the current directory. If the link is a union link, then the component name may be null. If filters or other information must be added, the MODIFY-LINK command should be used once the link has been created.

D.1.10 DELETE-LINK <options> <component> (opt: NUMBER <n>)

This command is used to remove an entry from a directory. Depending on the value of the LINK-EXP, this may or may not actually delete the entry. If it does not, the entry is flagged for deletion once the link expires. The possible values for <options> are shown in table D.1.

The optional NUMBER flag can be followed by the number of the link to be removed. This is necessary if there are multiple links with the same name; it allows the desired link to be specified. NUMBER is only valid with the LINK and ULINK options.

Table D.1: Options for the delete command

Option	Meaning
VLINK	Virtual link (either conventional or union) with given component name will be removed.
LINK	Same as VLINK, but only conventional links are checked.
ULINK	Same as VLINK, but only union links are checked
ANYLINK	Link with given name will be removed, even if it is in the native directory. If so, the contents of the file may be lost, and other links to the file may become dangling references.

D.1.11 MODIFY-LINK <component> <attribute> <optional-type-modification> <value>

This command modifies information associated with a link. <attribute> is the attribute to be modified and includes FILTER, DEST-EXP, LINK-EXP, HOST-TYPE, HOST-NAME, ID-TYPE, and FILEID. <optional-type-modification> may be ADD, DELETE, or

REPLACE for those attributes taking multiple values. <value> is the new value of the specified attribute.

D.1.12 MODIFY-ACL <options> <component> <entry-type> <auth-type> <rights> <principals>

This command is used to modify an access control list for a directory, or for a link within a directory. The directory is specified in the previous DIRECTORY command. The component is null (but quoted) if modifying the ACL for the directory (option= DIRECTORY). Options indicate whether the ACL to be changed is for a DIRECTORY, or a LINK. Another option indicates the operations to be performed (one of ADD, SUBTRACT, INSERT, DELETE, SET or DEFAULT), and whether to override the automatic inclusion of the system ACL (NOSYSTEM), or administer access for the client (NOSELF).

D.1.13 CREATE-OBJECT <options> <type> <component> ATTRIBUTE <attributes>

This command will create an object with the specified name and will return the identifier that can be used to open it. If a file, it is created empty. <attributes> allows the creator of a file to specify many of the attributes of the object at creation time. The form for each attribute would be the same as in EDIT-OBJECT-INFO.

D.1.14 CREATE-DIRECTORY <options> <component> ATTRIBUTE <attributes>

This command will create a directory with the specified name. The directory is created empty. <attributes> allows the creator of a directory to specify many of the attributes of the directory at creation time. The form for each attribute would be the same as in EDIT-OBJECT-INFO.

<options> tells the directory server what else is to be done when the directory is created. If options is VIRTUAL, then <component> is a name relative to the current directory, and a link to the new directory is added to the current directory. If <options> is PHYSICAL, then component is an absolute pathname relative to the root, and no link to the new directory is created. It is up to the application to add the link.

The access control list for the new directory will be a copy of the access control list for its parent. An entry will be automatically added to the ACL granting its creator all rights. If the LPRIV option is specified, only those rights needed to allow the creator to set up the directory (list, read, insert, and administer) will be added, and only if the creator does not already have such rights for the parent directory.

D.1.15 UPDATE <options> COMPONENTS <optional-component>

This command tells the server to check each of the named components in the current directory for forwarding pointers. If the referenced object has moved, the link will be updated. If <optional-components> is empty, all components in the directory will be checked. On success, the UPDATE command returns the number of links that were modified.

```
UPDATED <number> links
```

D.1.16 STATUS

This command requests the current status of the server. A humanly readable multiple line response is returned. The response may be presented to the user without additional processing. The response must conform to the following requirements so that it may be read by a program if desired.

The first line must include the server's software version identification enclosed in parenthesis, and the host name of the server. The name of the host should be the name that appears on local links generated by the server, it might not be the primary name of the host. The version identifier must be the first string that appears in parenthesis, and the host name must be the string that immediately follows the version identifier.

If a line contains a colon (:), the string preceding the colon identifies the meaning of the text that follows the colon. Reserved identifiers include Contact, Started, Memory, Data, Root, AFTP, AFS, and DB. The identifiers are case insensitive. If present, AFTP identifies the part of the file system accessible by anonymous FTP.

Other than for the first line of the response, implementations are free to add or modify lines that do not contain a colons. A sample status response follows:

```
Prospero server (Beta.4.2B) JUNE.CS.WASHINGTON.EDU
Requests since startup 4096 (3609+377+2 67+29+0 9 0+0+0 0 3) OF
Started: 26-Aug-91 15:14:56 UTC
Contact: bcn@cs.washington.edu
 Memory: 0(118)vl 0(4)at 0(5)acl 0(1)fi 1(1)pr 2(2)pt 0(711)str
   Data: /u1/vfs/pfsdat
   Root: /
   AFTP: /homes/june/ftp
```

D.2 Standard Responses

D.2.1 SUCCESS <identifying-info>

A command that does not generate any output returns this response if successful.

D.2.2 FORWARDED <host-type> <host-name> <id-type> <object-id> <version> <magic-no>

This response is returned when the object or directory that is the target of an operation has moved. The client can retry the response using the corrected information.

D.2.3 ERROR <text>

This response is returned to indicate an error encountered when parsing the request. The error might be a protocol error, or it might be the result of the server's inability to recognize a keyword or data type. <text> describes the error.

D.2.4 FAILURE <identifying-info> <text>

This response is returned when an operation can not be performed. The defined values for <identifying-info> appear below. <text> is optional text that provides additional information about the failure.

```
NOT-FOUND [FILE,ACL]
ALREADY-EXISTS
NAME-CONFLICT
AUTHENTICATION-REQUIRED
NOT-A-DIRECTORY
NOT-AUTHORIZED
SERVER-FAILED
```

D.2.5 WARNING <identifying-info> <text>

This response is returned to indicate a warning condition which does not affect the correctness of the response. This message can be used to indicate that the client is using an old version of the protocol that, while supported, should be phased out. It can also be used to inform the client of future changes on the server or scheduled downtime. The defined values for <identifying-info> appear below. <text> is optional text that provides additional information about the warning.

```
OUT-OF-DATE
MESSAGE
```

Appendix E

Reliable Datagram Protocol

This appendix describes the reliable datagram protocol (RDP) used by Prospero. As used by Prospero, this protocol is layered on top of the Internet User Datagram Protocol [Postel 80].

Prospero implements its own RDP because I was unable to find any that were suitable for general use. Most systems that use an RDP implement their own around the specific needs of their application. Like these other systems, early versions of the Prospero protocol defined the mechanisms needed for retries and packet sequencing. As these mechanisms were refined, the functionality was moved to a separate protocol layer to improve modularity, and in hope that this general and simple RDP can be used for other purposes.

This RDP was designed so that in the common case, the additional overhead of guaranteeing reliability is as small as possible. Unless special processing is required, the header is kept small, and unless a packet is lost, no additional packets are sent. If a field is not specified, the default value is used in its place. All fields up to and including the last field specified must be filled in, but the header may be truncated at any point from which all remaining fields are the default values. The RDP header contains fields described on the next two pages.

Bytes 0	Version and header length: High order two bits are RDP version number mod 4 (this is version 0). Low order six bits are the header length including byte 0.
Bytes 1-2	Connection ID: Defaults to the expected connection ID. A specified value of 0 means use the default.
Bytes 3-4	Packet number: Defaults to 1 if not specified. A specified value of 0 indicates an unsequenced control packet which should not be passed to the application.
Bytes 5-6	Total number of packets: Defaults to 0 if not known, or retains current value if it was provided in any earlier messages. If the packet number was also not specified, then it defaults to 1. A specified value of 0 means use the default.
Bytes 7-8	Received through: Total number of packets successfully received by peer. Defaults to current value if specified in previous message. Defaults to 0 otherwise. If a value of 0 is specified, then it means reset to 0 (i.e. it forgot the earlier messages).
Bytes 9-10	Backoff (expected time to next packet): Defaults to current value. Specified value of 0 means use default. This is a hint. The client can use a different timeout if it has reason to believe messages are available which have been missed (e.g., gaps in the list of received packets). In any case, the client is always free to use a timeout longer than that specified here.

Bytes 11-12	Flags: Byte 11 is a bit vector, the high order bit (7) meaning please acknowledge and the next bit (6) indicating that the packet is a sequenced control packet only and should not be returned to the application by the rdgram library. The low order bit (0) means that a protocol ID for a higher level protocol follows. The next bit (1) means that a 2 octet priority is specified. Byte 12 specifies 1 of up to 256 other flags. The flags may themselves require additional fields specific to the flag. These fields appear at the end of the header in the order they are needed when reading flags from the low order bit to the high order bit, followed by any extra fields needed by the flag specified by the 12th byte. A value of 255 for the 12th byte is reserved for future expansion.
Bytes 13 and above	Fields specific to particular flags
Next 2 bytes	Protocol ID (if protocol id flag specified): These bytes identify the interpretation of the data carried in the packet. The default and a value of 0 mean that it is not specified, but has been agreed upon externally (i.e. the applications know).
Next 2 bytes	Priority (if priority flag specified): These bytes are a signed integer representing the priority of the request. Not all implementations understand this message, and many that do will not honor requests for expedited handling. Negative numbers indicate expedited handling while higher numbers indicate greater delays. A priority of 0 is normal.

Vita

Barry Clifford Neuman was born in New York City on September 27, 1963, and grew up in Orange County, New York (near West Point) where he graduated from Cornwall Central High School in 1981. He received his Bachelor of Science from the Massachusetts Institute of Technology in 1985 and worked for MIT's Project Athena for the next year. In 1986 he began graduate studies at the University of Washington, receiving the Master of Science degree in 1988 and completing his doctoral studies in 1992.