Representing Information about Files

by

Jeffrey Clifford Mogul

Department of Computer Science

Stanford University
Stanford, CA 94305

REPRESENTING INFORMATION ABOUT FILES

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Jeffrey Clifford Mogul
March, 1986

Abstract

Intelligent use of files depends on having sufficient knowledge about them: their purposes, structures, and contexts. Humans have traditionally made do by using their own methods for capturing and manipulating such knowledge, but this is not available to programs, nor is it necessarily convenient for humans. Computer file systems must make this knowledge more widely available, by formalizing its storage and providing efficient manipulations.

Explicit and careful treatment of knowledge about files makes possible the effective solution of otherwise intractable problems. These include the sharing of files in heterogeneous distributed systems, and the location of files not by path names but by predicates on their properties.

The thesis proposes a model of files that includes the concept of file properties. It then discusses how file properties are used in systems with inadequate property support, and how they could be used in a system that treats them explicitly. Interface specifications are presented for a file system and a directory system based on the model, and it is shown how these systems can be efficiently implemented. Performance measurements demonstrate that such systems are not significantly slower than traditional systems, and are dramatically faster for certain functions.

Key words and phrases: File systems, directory systems, file properties, database applications, database implementation, heterogeneous distributed systems, file search performance, directory performance.

***Computing Reviews* categories:** E.5, D.4.3, H.2.5, H.3.3, C.2.4, D.4.8.

Acknowledgements

By convention, this section of a thesis begins with "I am indebted to my thesis advisor Professor ——, without whose guidance, support, and tireless reading of innumerable drafts this thesis would not have achieved its final form." All true. My advisor, Professor Brian Reid, read the innumerable drafts carefully and critically, if not always tirelessly. He read them with his red pen uncapped and ready to pounce on bad spellings, bad grammar, bad style, and bad science. He let me argue with him until we agreed about the science; if I won arguments I should not have won, that fault is mine. Brian's support was complete; he stood up for me on matters large and small, and never stopped believing in the thesis.

The other members of my committee, with their different points of view, provided advice and skepticism that undoubtedly strengthened the thesis. John Hennessy toted a draft around Europe, saving for me the pages he marked up and leaving the rest in waste bins far and wide. Forest Baskett as well gave me detailed comments, with nanos to spare; he and John have been keeping me honest about computer systems since I arrived at Stanford.

In addition to Brian, John, and Forest, I had a shadow committee to give me comments, advice, and support. Keith Lantz and David Cheriton, in particular, had a significant influence on the direction I followed. Students and staff of their research group provided both a sounding board for my ideas, and a workstation environment that made my life a lot easier.

As a graduate student, I had my jousts with arcane workings of Stanford. I was rescued from the machinery by many dedicated staff members; Carolyn Tajnai and Victoria Cheadle stand out for their good humor and skillful assistance. Linda Kovach managed to get me appointments with Brian on a regular basis, in spite of considerable competition for his time.

Finally, I must thank my friends for supporting and tolerating me over the years. I cannot list all my housemates past and present, but Beth Newell deserves special mention, and not just because she has had to put up with me longer than any one else did. Neither could I list all my other friends, but I owe special thanks to Marianne Winslett, Amy Lansky and Steve Rubin, Linda Gass, and Connie Humphries. The rest of you know who you are.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

This thesis is about better file systems.

Better file systems allow us to manage our files more effectively, solve problems that cannot now be efficiently solved, and build better software.

There is more than one way to fulfill each of these goals. Building a better file system is a way to fulfill all of them. A file system design that includes the right primitives is the most effective solution.

Intelligent use of files depends on having sufficient knowledge about them: their purposes, structures, and contexts. Humans have traditionally made do by using their own methods for capturing and manipulating such knowledge, but this is not available to programs, nor is it necessarily convenient for humans. Computer file systems must make this knowledge more widely available, by formalizing its storage and providing efficient manipulations. By doing so, they will contribute to effective solution of otherwise intractable problems.

1.1. A file is more than its contents

A general purpose computer without a file system is a toy. A computer is used to manipulate data, and for any significant application some of this data is worth saving. A file system is where we store a lot of data for a long time. Some people prefer to store all their data in a data base system. This thesis is directed to the majority who prefer file systems.

When we talk about what is "in" a file, we refer to its contents. This is not the only information associated with a file: file systems store information *about* a file as well as in it, and this information is of vital interest to users.

We call this information about a file its *properties*. Some systems use other terms, such as "attributes" or "status." For example, the properties of a file usually include its length, owner, and date last modified.

Although the quantity of information stored as file properties may be insignificant next to that stored as file contents, it attracts significant attention. The directory listing command, in a study of the relative frequencies of Unix command invocations [Ritc78], was second only to the text editor, accounting for almost 10% of the commands. Clearly, users are interested in the properties of their files.

Programs are even more interested in file properties. Roughly half of all Unix command programs use the system call that obtains the properties of a file. Our measurements, presented in section 8.1, of a heavily used Unix system indicate that this system call is used more than twice as often as that which opens a file for reading or writing.

The properties of a file include the information the file system "knows" about the file.

However, a file often has properties meaningful only to the client, and few file systems allow the definition of new properties. Without *extensibility*, a client cannot use a file system's property mechanism to store properties that the file system designer did not anticipate.

1.2. File systems are for sharing

A file system is more than just a place to store data for a while. It is a place to store data so that it may be shared by various users of the computer system. Even with the proliferation of decentralized personal computers, we still use centralized file systems to promote sharing. Because files are the fundamental focus of long-term sharing in a computer system, a file system is the "soul" of the computer system in which it is embedded; it must provide the continuity of sharing across both time and space.

A file system reflects a long history of sharing, often beginning long before the computer was built. The compiler that you use on a newly-installed computer might have been written years before; the copy of it stored on your file system is a way of sharing it with the programmers who wrote it. Any file system shared by a community of users accumulates files representing a collective history of the community.

In a distributed system, files are shared across "space" as well as time. Distributed systems often arise not from a coordinated plan, but by connecting existing computer systems in ways not anticipated when they were originally designed. Even carefully-designed distributed systems will be connected to other systems over long-haul networks. We can expect almost any system to be incorporated, in this incremental way, into a loosely-coupled distributed system. Inevitably, such systems are *heterogeneous*: not every node uses the same hardware and software. It is important, nonetheless, to be able to share files between the nodes.

Because programs and users are so interested in file properties, sharing a file means not just sharing what is in it; it means sharing what we know about it: sharing a file means sharing its properties. In a heterogeneous distributed system, this is not always easy. Further, the hardest part of porting a program to a different system often is converting the code that manipulates file properties. While access to file data is becoming *uniform* across different computer systems, access to properties is not.

Besides sharing files over time and across different system implementations, we would like to share programs in the same way. In other words, we would like our programs to be portable; but to write portable programs, we need a *uniform* file system interface.

Our file systems must support sharing. To do this, they must not only provide a consistent "read/write" interface, they must also support file properties explicitly, extensibly, and uniformly. Uniformity means commonality of the file system interface, over time and between systems in a heterogeneous network. It requires an underlying model of files that is shared by all systems in the network. This thesis proposes such a model.

1.3. Better file systems support file properties

A file system can only be called "better" if it solves problems that current systems can't. Explicit support for file properties does solve a variety of problems. This thesis examines the problems that arise with current systems, and shows how systems based on our model allow efficient solution of these problems.

A model is not a file system. This thesis specifies the design of a file system, and a directory system, both of which are based on our model. These system designs are detailed interfaces, allowing a variety of implementations.

Efficiency is essential. This thesis describes several approaches to implementing the designs presented. Measurements of these implementations suggest that a domain-specific approach is superior to use of a general-purpose database, and that effective support of file properties is not unpleasantly expensive.

Efficiency does not come merely from a good implementation. Our model, by providing a new approach to file properties, enables efficient solutions to a large class of problems where before only inefficient solutions were possible. Although it is theoretically possible to solve many of these problems with existing systems, in practice these solutions are unacceptably inefficient. An inefficient solution is not a solution.

What is important about this thesis is not that our implementation is fast, but that a new paradigm, explicit treatment of file properties, allows us to consider efficient solutions where before this could not have been done.

1.4. Road map to this thesis

The thesis is divided into three parts. The first part provides motivation and background for the subsequent parts. It begins with chapter 2, a summary of the major ideas in the thesis. Subsequent chapters restate these ideas in greater detail.

Chapter 3 discusses previous work in the area. Although no one has taken the comprehensive approach of this thesis, several existing systems support file properties to some extent. There has also been some work on sharing of properties in a heterogeneous environment.

The second part describes the structural ideas of the thesis. Chapter 4 presents our model of files, directories, and file properties. Chapter 5 shows how properties are used in current systems, and discusses the problems arising from inadequate support.

Chapter 6 presents designs for a file service and a directory service based on this model. Chapter 7 describes several implementations of the directory service, and draws some conclusions as to how to achieve good performance.

Chapter 8 presents performance measurements for our implementations and for their application to actual problems. It shows how performance improvements come primarily from our new paradigm, rather than from its efficient implementation.

The third part looks at the prospects for applying the ideas of the thesis to the solution of various interesting problems. Chapter 9 shows how improved file property support creates more efficient solutions to real problems. In Chapter 10, we consider the particular problems of file properties in heterogeneous distributed systems.

Chapter 11 points out unsolved problems and areas for further work. Finally, chapter 12 summarizes the conclusions to be drawn from this research.

Part I

Motivation

Introduction to Part I

The two chapters in this part are meant to provide motivation for the material in subsequent chapters of the thesis. There is no single theorem, algorithm, or observation that catches the gist of this thesis; rather, the point is that a wide variety of problems in computer systems design may be solved only by understanding file properties.

Just as there is no simple problem here, there is no simple solution. Rather, the solution is a combination of paradigm, system design, efficient implementation, and creative application of these features.

For these reasons, the best motivations for the ideas in this thesis are each other; the problems motivate the solutions, while the solutions illuminate the problems. Chapter 2 describes, in broad terms, these problems and solutions.

It is a fact of Newtonian physics that astronomical bodies orbit not about each other, but about their combined center of mass. The value of this thesis is that it recognizes the constellation of problems and solutions orbiting around the concept of file properties; even though there may not be any *thing* there at the center, that center is real.

Still, the lack of an observable thing that is file properties may explain why until now nobody has described this constellation. Chapter 3 covers a variety of relevant approaches to files, properties, and their applications. Many of the individual ideas in this thesis have been dealt with before; there have even been some gropings towards a coherent understanding of file properties. There is no evidence that such an understanding has ever been reached.

Chapter 2
Overview

This chapter is a summary and overview of the thesis. It introduces all the major ideas, and so serves as motivation and context for the following chapters. Even if you intend to read the entire thesis, you should first read this chapter.

Because this is an overview, the ideas presented are not much buttressed with justifications, references, figures, or tables. These are included in the detailed chapters, as indicated by cross-references in this chapter.

2.1. A new approach is needed

When computer file systems were first developed, they provided a simple mechanism for long-term storage [Corb63, Wilk64]. A file contained a sequence of logically associated data, such as a program binary, or a document, and could usually be given a name suggestive of its contents. Successive improvements in file system concepts led to structured directories [Dale65] which allowed logically related files to be grouped together. However, the ability to represent information *about* a file (as opposed to its contents) usually has been limited to such name-borne information, and perhaps length, protection, ownership, and access times. Occasionally a limited type facility is available ("text," "executable," etc.) The user's interface to this information has seldom been uniform within a system, let alone across different systems. Instead, the user is faced with a motley assortment of bit fields, flags, and integers, and has difficulty encoding information not anticipated by the file system designer.

We assert that this is an unsatisfactory situation. The lack of a uniform interface abstraction for file properties, support for manipulating them, or indeed any apparent consideration of properties in the design of most file systems, leads to poor solutions of a variety of problems. These inappropriate solutions are inefficient, incomplete, fragile, and cumbersome. What we need is a simple, efficient system upon which we can build good solutions.

We present a new approach to the problem of representing information about files. Section 2.2 illustrates the need for a more coherent approach. Section 2.3 discusses the requirements for complete support of file properties. Section 2.4 presents a set of useful data abstractions, based on the Lisp idea *of & property list,* that are then used in section 2.5 to construct directory and file system designs that allow appropriate solutions to our problems. Section 2.6 describes an experimental implementation of a directory service based on the property list representation, and touches on issues of efficiency and reliability. Finally, section 2.7 summarizes and discusses the significance of the ideas in this chapter.

2.1.1. Environment

Throughout this thesis, we address systems that are part of a loosely coupled distributed system. We will assume that the environment contains a number of client processes interacting with a number of server processes using remote procedure call. One can consider a single-processor system to be a degenerate case of this model.

2.2. Problems Addressed

We start by looking at three simple examples of situations that arise in the use of file systems:

- **What is this file?** We may store files of many different types; a single file might be of more than one type. For example, this thesis is stored in a file that, conceptually, has the types:
  - Text
  - Scribe-Input
  - English-Language
  - American-Spelling
  - Thesis-Format

  each of which may be of interest to tools such as a spelling checker. We can't expect the designer of the file system to anticipate all the possible types that we might want to assign; how can we mark a file with all its types?

- **File exchange in a** heterogeneous **distributed** system: Many organizations have local networks connecting a heterogeneous collection of workstations and file servers. We want programs on the workstations to be able to access files on all of the file systems. It is not hard to arrange for programs to read and write files in a heterogeneous environment, but can a program determine the length of a file, or when it was last modified?

- **Searching for a file:** A new computer arrives, and we want to install system software on it. One way to do this is to copy all the files from an existing system. If we do this, for example, with a 4.2BSD Unix system, about 1% of the copied files are system-specific and must be changed for the new system. How can we find these files?

These problems all arise because using computer files means using information *about* them as well as information *in* them.

All file systems store information about files; few do it well. The three fundamental problems with current systems are:

1. Useful or necessary information is not available, except as "expert" knowledge on the part of users.
2. Such information as is stored is not available in a uniform manner, so programs that use it are hard to write and are not portable. Sharing files in a heterogeneous distributed system is impractical.
3. It is difficult to locate files specified by their properties, rather than by their names; the search mechanisms available are woefully inefficient.

A lack of information about files can cause problems when files are used for sharing, instead of simply as stable storage for single applications. Sharing can arise in several ways: a file

might be input for a different program than that which generated it; a file might be moved (migrated) from on file system to another; a file might be shared by several users; the file system and its clients may share information about the file.

In order to share information about a file, there must be a way to to associate such information with the file, and all potential users must have access to it. Current systems provide neither adequate facilities for associating information with files, nor uniform methods for accessing the information.

We also need efficient ways to search for files. Although it is easy to find a file knowing its name, users often need to find a file knowing only something about its properties. In effect, the properties of all the files stored by a file system form a database, yet current systems provide only excruciatingly inefficient ways to search this database, and so many potential applications are impractical.

Many seemingly unrelated problems in computer systems are actually symptoms of this larger problem, the inability of current file systems to support file properties. Some specific examples:

- Interchange of files between dissimilar systems is made easier if the "type" of the file is known. For example, to transfer a file containing an array of binary numbers between machines with different byte-orderings requires a transformation different from that required to transfer a text file.
- Different systems use different formats to represent similar information. For example, most modern file systems allow the specification of protection information for a file, but there is little consistency in exactly what "protection" means. Moreover, the presentation of such information varies considerably from system to system.
- Automatic migration of files between two servers (perhaps from a slow, cheap server to a fast, expensive one when the file becomes "active") is impeded by non-uniform representation of file attributes.
- Changing the underlying representation for a file system is harder than it should be; with a static representation, it might be easy to convert the stored information but hard to convert the programs that use it.
- Many files are related in some way to other files. For example, a program object file may depend on several source files. These dependencies can be represented in an auxiliary file [Feld79b, Schm82], but maintaining this file is a problem. Relationships may also be indicated by similar file names, or by grouping related files in directories. While these methods are helpful, they are neither robust nor sufficient.
- When a subroutine library is updated, it is often important to find and recompile the programs that depend on it. To do this by iterating over the entire file system, checking each file, is quite inefficient; it should be possible to efficiently locate the files that are relevant.

We believe that these and many other problems are manifestations of a single, deeper problem: there is a need for a coherent approach to information *about* files.

This thesis argues that to solve these problems, and many others, file systems should provide three facilities:

1. An *extensible* mechanism for storing information about files, so that file system clients can associate all necessary information with their files.

2. A *uniform* interface to this information, so that it can be shared by all users of a heterogeneous distributed system, so that all applications can share the information, and so that programs that use the information are portable.

3. An efficient *search* facility, to allow users to locate files based on knowledge of their properties.

These facilities must be part of the file system, not an independent service or a feature of certain applications. It is always possible in principle to simulate file property support using existing file system mechanisms, but one cannot simulate good performance.

2.3. Requirements

A coherent approach to file properties requires attention to three areas:

- What information can be stored?
- Where is it stored?
- How can clients manipulate the information?

In this section, we answer these questions.

2.3.1. What Information Should be Stored?

Traditionally, file systems store information necessary in managing files. This usually includes file length, ownership, protection, and access time. We call these *intrinsic* properties, since they are central to the operation of the file system.

In addition, some systems allow the specification of a restricted set of types. These are often limited to those file types that the system must know about. For example, Unix keeps track of which files are directories, executable, and "special." Some systems store a limited set of non-intrinsic properties; for instance, the CMU CFS [Acce80] includes a user-defined "advisory file type."

No predefined list of properties can be exhaustive. As users create new applications, they inevitably need to represent new kinds of information about their files. Moreover, this information often cannot be represented in a fixed format such as a bit or bit-field. Instead, it is usually of some simple data type such as a number, a short character string, or a boolean flag. It should be possible to store arbitrary information of one of these types, and to give the properties arbitrary names. We call these *non-intrinsic* properties, and file systems that support them are *extensible.*

2.3.2. Where should information be stored?

There are three basic ways of associating information with a file:

1. *Internally*, as part of the contents of the file,

2. *Directly* attached to the file (but not part of the contents),

3. *Indirectly*, in some object that refers to the file.

The common practice of encoding information in a file name can be useful, but this is of limited capacity and should only be used as a "hint." We need some place besides the file name if we

are to store arbitrary information. We will see that both direct and indirect association are necessary, and internal association is incapable of meeting our goals of uniformity and extensibility.

Although many existing applications establish conventions for storing property-like information internally to files, these conventions are neither uniform nor universal, and so only small cliques of applications can communicate in this way. Application-defined conventions are insufficient to store arbitrary information; they are not extensible. In particular, a user may wish to annotate a file, even if the file does not contain text; thus, storing the annotation internally is not always possible. For example, a crash dump might be marked "This crash occurred when we ran the PL/I compiler." It is unreasonable to expect the application designer to establish a convention for attaching arbitrary properties in files such as crash dumps.

Direct association means that properties are firmly attached to their files. A client that needs access to the properties of a file should be able to use a simple, consistent mechanism that is independent of the particular application or file format. This requires that access to such properties be mediated by some universally available layer. Since intrinsic properties are in any case mediated by the file system that manages the file, it makes sense that the file system provides access to all directly-associated properties.

The most common form of indirect reference to a file is a directory entry. Directory entries always store information about files, but existing systems vary in how much information they attach to a directory entry. For example, Unix directory entries are simply bindings between a name and an internal identifier. Other systems store most information about a file with its directory entry; such systems require a one-to-one mapping between files and directory entries.

Systems without this one-to-one restriction are more flexible. Different users may want to refer to a single file in different naming environments, or by completely different names. Thus, users should not be forced to store file properties solely in directory entries. It should always be possible to store properties directly with the file, to avoid the potential inconsistencies associated with multiple copies of information.

On the other hand, there are some kinds of information that apply to a directory entry instead of to the file that it denotes. For example, it is often interesting to know who made an entry, and when; it may also be important to make a distinction between the right to delete a file, and the right to delete a particular directory entry for it. These properties really should be stored in the directory entry, not directly with the file.

We can also use a directory entry as a *cache* for properties stored with the file. In this case, the "true" information is associated with the file, but copies are also stored in the directory entry. This allows users to efficiently search a directory for files meeting some criterion. Guaranteeing consistency of these caches in a distributed system probably requires more resources than might be saved when the caches are used. However, with many applications the reality is that some properties can be safely cached even without automatic consistency maintenance, because their values are not expected to change. That is, their consistency depends on their semantics, not on a mechanism supplied by the file and directory system. One such application is described in section 9.1.1.

We conclude that information about files should be stored in two places:

1. Directly associated with the file, with access mediated by the file system
2. Indirectly via directory entries, but with the understanding that the consistency of cached values is not guaranteed.

2.3.3. How Can Properties be Manipulated?

Our goal of providing a uniform interface to file properties requires us to consider what functionality must be provided; that is, how clients can manipulate properties. Here we sketch what we consider to be the necessary primitive operations.

We can divide the operations into two classes: simple operations, involving specific single properties or files, and search operations, where we use predicates on property values to find a set of files. Existing systems provide only simple operations.

The simple operations can in turn be broken down into implicit and explicit access to properties. Implicit access is performed by a file server in the process of executing a file operation. For example, when a client issues a write operation, the "length" and "modification time" properties must be updated; implicit operations apply only to intrinsic properties. Since one of our goals is to provide an extensible set of properties, we must provide explicit operations not only to read and write properties (as found in traditional systems) but to create and delete properties as well. Also, since the set of properties that a file has cannot be known in advance, we must provide a mechanism for clients to discover just what properties have been associated with a file.

Search operations can be characterized by the domain of the search, and by the allowable predicates. In each dimension, the choices are a compromise between what clients desire and what can be implemented. In addition to "the entire file system," we might want to provide both smaller domains, such as the set of files listed in one directory, and larger domains, such as all the file servers in a distributed system. We will show (in chapter 9) that "entire file system," or *global* searches are both useful and relatively easy to implement. Searches focussed on a single directory are also clearly useful, since they limit the number of irrelevant responses and improve performance by reducing the search space. The most efficient approach to searches involving multiple file servers is probably to provide efficient global search support for single file system searches, and perform these in parallel on each server; we do not further address domains larger than an entire file system.

We also need to decide what kinds of query predicates to support. At the least, we need to be able to find files with properties that match a specific value; for some properties, such as those with numeric values, we should be able to search for values lying in a specific range. On the other hand, while one can conceive of more elaborate predicates, it is not clear that they can be implemented efficiently. For example, a client might want to search for string-valued properties matching a pattern, but it is not clear that the space and time costs of supporting efficient pattern-based string queries are repaid by better query performance; see section 11.2 for more discussion.

2.4. Important Data Abstractions

In this section, we propose a set of data abstractions useful in representing information about files. Since these are abstractions, rather than physical schemes, we define them as sets of operations on abstract data types. In effect, we are defining a syntax for manipulating file properties; in this section, we are not concerned with semantics.

In chapter 4, these abstractions are presented and justified as part of a comprehensive model of files. Here, we simply introduce them as given, and trust the underlying concepts to the reader's intuition.

2.4.1. Properties

The lowest level data abstraction is the *property*. A property has a name and a value. Property names are human-sensible character strings, so we can always fall back on human intelligence if our applications are not powerful enough. Property values are (*type*, *simple-value*, *protection*) triples.

The type of a property value indicates something about its meaning, and hence the operations one can perform on it. For example, the "Length" property of a file has an integer *simple-value*; one might want to do arithmetic on it. By contrast, the "Owner" property may have a string value; it is not appropriate to do arithmetic on it, but one might want to use symbolic names for owners. Other reasonable types for properties include boolean, real, and date/time values. An arbitrary type structure is neither necessary nor desirable, since it quickly complicates the interface. Properties have "small" values, and unusual types can be encoded as strings.

In our model, a property has protection information associated with it. Associating protection information with each property, rather than simply with a file as a whole, is necessary to avoid the problems associated with "all or nothing" protection schemes. For example, the owner of a file might wish to grant others the right to modify its contents, but not its protection.

2.4.2. Property Lists

Objects usually have more than one property, and it is convenient to have an abstraction that allows us to manipulate sets of properties. This abstraction is reminiscent of the Lisp *property-list*, and so we use that term. In Lisp, a property list is associated with each atom, and is effectively a list of (*name*, *value*) pairs. In our representation, a property list has a name (rather than being bound to an atom), and is a set of *name* ⇒ *property* bindings; no further substructure is imposed.[1] We allow a property list to contain arbitrarily many properties.

The basic operations on property lists include operations to create and delete property lists. There are also operations on single properties:

PutProp (PropertyListName, PropName, PropType, PropValue, PropProtection) ⇒ (Status)
Inserts a new value binding for the property name; if a binding already exists, it is replaced.

GetProp (PropertyListName, PropName) ⇒ (Status, PropType, PropValue, PropProtection)
Returns the value currently bound to the property name, if one exists.

RemProp (PropertyListName, PropName) ⇒ (Status)
Removes the binding associated with the property name.

We provide a pair of operations to iterate over all of the properties on a list. To avoid storing client-dependent state at the server, the operations encode the iteration state in a value (the *IteratorKey* operand) that is passed back and forth between client and server. An iterator key value denotes a specific property on the list, and has a unique successor. Once obtained, it may be used even after its successors. We can lock the property list to ensure that it is not modified by one client while another is iterating over it.

InitIterator (PropertyListName) ⇒ (IteratorKey, Status)
Initializes an iterator.

---

[1] Property lists were introduced into Lisp for a similar reason. According to John McCarthy [McCa82], they were seen as a way of storing information about objects that could not be inferred from their structure. The structure of Lisp objects comes from the relationship between atoms, while the structure of file objects is generally internal, but the problem is more or less the same. We use the Lisp notation to describe specific instances of property lists. However, it is important to realize that no ordering should be inferred from the notation.

IteratorNext (PropertyListName, IteratorKey) ⇒
(Status, NewIteratorKey, PropName, PropType, PropValue, PropProtection)
Given an iterator key, returns the name and value of "next" property on the list, and a new iterator key which may be used for the following iteration. The client should not assume that the properties will be produced in any fixed order.

In addition to operations on single properties, we have query operations that refer to a large set of files; for example, one might want to list all of the files on a system that have not been backed up since they were last modified. A query operation takes the property name and a predicate to apply, and returns a list of (*PropertyListName*, *PropValue*) pairs via the iterator mechanism described above. We support equality and inequality predicates on all property types, and subrange predicates on types where this can implemented efficiently.

All of these operations are connectionless and obey "at-least-once" semantics (as defined by Spector [Spec82]). These features are attractive in a distributed system. Connectionless operations interrupted by a crash will be able to proceed upon recovery without having to recreate a broken connection. At-least-once semantics ensure that repeating operations (because of a crash or communications failure) will not cause problems.

2.4.3. Property List Groups

In our system designs, property lists are always gathered into *property list groups*, another abstract data type. A property list group is simply a named set of named property lists; it is a name space for property lists, thus providing structured names. Another benefit of the structure provided by grouping property lists is that it provides an intermediate-size context for queries. Queries over property list groups include, for example, listing all of the text files in a directory, or finding which versions of a file were "current" during a certain week.

This abstraction includes all the operations on property lists, but qualified by the name of the property list group. It also includes operations for creating and destroying property list groups, for iterating over the names of the property lists in a group, and for performing queries within the context of a group.

2.5. System Designs

We will present, in chapter 6, practical designs for a directory system and a file system based on our model of files and file properties. The specificity of these designs is for the sake of concreteness; we do not mean to preclude other design options. In this section, we sketch the designs in order to introduce the important concepts without getting mired in extraneous detail.

2.5.1. Representing Directories

A directory is a set of *name* ⇒ *properties* bindings, or *entries*. In a conventional system, there is a fixed set of possible properties for an entry, and so a positional representation can be used. When the entries contain arbitrary properties, an extensible representation, such as property lists, is necessary.

We represent a directory as a property list group. Each named property list in the group represents an entry. A consistent and useful directory system needs some further structure imposed, so access to a directory is mediated by a directory server. The server maintains standard properties for each entry, among them:

- **Entry Type:** The type of the directory entry. In order to implement a directed graph of directory nodes, this property is used to distinguish leaf nodes from other directory nodes. The entry type allows a client or server to decide how to interpret an entry.
- **Lower-level Name:** A name for the denoted object, understood by the server that manages it. This might be a globally-unique identifier for a file.
- **Server Name:** A name for the server which manages the denoted object, suitable for use by the communication mechanism.

From this simple mechanism, one can construct a variety of directory structures. For example, one could use a symbolic link

```
((Type file-name)
        (LowerLevelName /usr/bin/emacs)
        (Server SomeUnixSystem))
```

or a "hard" link

```
((Type inode-number)
        (LowerLevelName 43346)
        (Server SomeUnixSystem))
```

Notice that the denoted object can be managed by a server at any site.

A directory service maintains and understands a variety of other intrinsic entry properties, including access times, protection, and ownership. A client may also maintain information with the directory entry. As mentioned in section 2.3, it may "cache" copies of certain properties stored with the object itself, in order to speed subject-related searches of a directory. For example, the type of the underlying object might be stored as part of the directory entry, since it is unlikely to change, and since it allows a client to search for objects of a given type while interacting only with the directory server.

2.5.2. File System Design

A key feature of our file system design is that it explicitly supports multiple versions. We begin with a brief digression to argue that this is necessary, and that one can properly support file properties only in a multi-version system.

Many files are not static; they are modified after they are created. One can identify distinct *versions*, or points in the history of a file after a set of related modifications. Program sources are one example of files for which the concept of version is useful. Unfortunately, some file systems maintain only the current version of a file; previous versions of interest must be explicitly rescued from deletion. A number of existing file systems (for example, TOPS-20 [Digi80], VAX/VMS [Digi78], and the Cedar CFS [Schr85]) do support *multi-version* files. They are thus capable of providing intelligent support based upon the presumed relationship between different versions of a file.

In systems that do not support multiple versions, two problems arise:

- *If a user wishes to store older versions of a file, the system supports no consistent way of naming these older versions.* The user must explicitly rename the current version when a new version is created. Often, it is necessary to omit a version specifier from the current version, because the software considers every character of the file name significant, and has no way of denoting "the current version" as opposed to some specific version.

- *The file system is unable to make use of any presumed relationship between the versions of a file.* Non-default properties of a file must be respecified as each new version is created, renaming a file requires the user to rename each version separately, and the system cannot make use of the inherent supercession of older versions to decide when to archive them.

The most unpleasant consequence of systems without version support is that old versions disappear when new versions are created, unless the user explicitly preserves them. A principal difference between computer files and the paper files they are replacing is that a paper file involves a history, but a computer file only reflects a current state. Users accustomed to paper files might prefer systems in which deletion of old versions is done explicitly, instead of implicitly. This becomes more acceptable as the cost of disk storage goes down, and the demand for user-friendly and forgiving systems increases.

We represent a multi-version file as a property list group. Storage for the contents of the file is provided at a lower level by a simple, single-version file system. The "name" of the property list group is the global unique identifier for the file; the names of the property lists are the identifiers of die versions of the file. The file, as an object, is the set of versions, and thus a directory entry does *not* refer to a specific version, but rather to all the versions of a file.

Since some properties of a file will be shared by all of its versions, there is a property list associated with the file as a whole, in addition to the lists associated with each version. Thus, properties expected to remain constant across all versions (such as the owner and type of the file) are put on the file's property list, while properties that are expected to change (such as access times and size) are put on the version property lists. If a property appears on both a version property list and the file property list, the more specific one (the one on the version list) takes precedence. In this way, we provide an *inheritance* mechanism that may be overridden for specific versions.

2.5.3. Global Queries

A file system normally contains a large number of files, Clients may want to construct "global" queries about all the files in a system; for example, "what files were modified last month?" or "what programs depend on the graphics package library?" While it is possible to answer such queries by looking at the properties of every file in the system, this is extremely inefficient

Instead, we provide a global query operation that is analogous to the query operation on property list groups, but with the entire file system as its domain. With an appropriate implementation, answering these queries takes time roughly proportional to the size of the answer, instead of the size of the file system. (We present more specific performance results in section 9.1.1.) This makes global queries far more practical and thus far more useful. Both our file system and our directory system support global queries.

2.5.4. Heterogeneous Distributed Systems

A heterogeneous distributed system presents two barriers to effective file sharing. First, it is difficult to remotely access a file on one kind of node from a program written for another kind of node, because the file system interfaces are different. Second, when a file is copied from one kind of node to another, it may not be possible to represent exactly the same properties.

Both of these problems arise because there is tremendous variation in the file system inter-

faces of existing systems. This variation is unnecessary; eliminating it would eliminate most of the barriers to sharing in a heterogeneous distributed system. Further, the variation in modern file system interfaces is usually in their approaches to file properties; there is little substantive variation in the way that file contents are treated.

The primary contribution of our approach is that it is uniform; it provides a model which, when followed in designing a file system, ensures that programs written for one such file system will work with any other. However, extensibility is also crucial in solving the file-copying problem; if we can store arbitrary properties, file copying can be inverted without any loss of information. The intermediate file system may not understand the additional properties, but it will not throw them away.

Although it is most useful for designing new systems, our model is helpful in integrating existing systems into a distributed system, since it provides an interface abstraction that can be at least partially mapped onto existing interfaces. It is simple to re-write an existing set of file system library routines to make a property-list-based system appear to be any traditional one; this allows use of unconverted existing programs in a newly-designed system. It is also possible, as we show in section 7.4, to write a "front end" that maps property list operations onto those provided by an existing file system, allowing new programs to be used with an old file system. Such a front-end will of course not be a complete implementation of the abstract interface, because the underlying system might not provide, for example, the ability to add new properties. However, for a large class of programs, the use of front-ends will provide sufficient access to existing systems, making it feasible to use them in a distributed environment.

2.5.5. Property naming and semantics

How do we establish the meanings of properties and agree on their names? The meaning of an intrinsic property is imposed by the file system, but when a user defines a new property, its meaning comes solely from the user's intention. One cannot determine the meaning of a property solely from its name and value.

In a large system, this leads to several problems:

- The same property name might be employed by different users to denote more than one meaning.
- Different users might use more than one property name to denote the same concept.
- A name assigned by one user might be meaningless to another.

The use of human-sensible names, such as "File_Expiration_Date," can reduce the ambiguity; it can never actually eliminate it. We need more formal mechanisms. This is a difficult problem; we discuss it in more detail in section 11.1.

2.6. Implementation

Our file system could be implemented as the client of a general-purpose database system, but probably not with acceptable efficiency. We believe that an efficient implementation requires a problem-domain-specific approach, combining strong assumptions about the use of the "database" with a realization that some plausible operations are not worth supporting. For example, we assume that there will be significant locality of reference, so that data about a specific file should be clustered together. We expect "join" queries to be relatively infrequent, and thus do not provide the complex mechanism required to optimize these at the server.

As a test bed for our model, we implemented the core of a directory system based on our design. A file system that supports multi-version files can be based on a similar implementation. We made several assumptions, based on informal studies of existing systems:

- The median number of entries in a directory is much less than 100, as we show in section 8.2; this may be because people cannot visually scan more file names than will fit on a page.
- The average directory entry will have a few dozen properties.
- Many property names and string values will be repeated often; a space-efficient scheme should avoid storing multiple copies of such data.

An earlier implementation used a different approach that ran afoul of these assumptions, and so wasted a lot of storage space.

The directory system uses a set of local databases, one for each directory. It also uses a global database to store property names and string values, to avoid storing multiple copies.[2]

Each local database stores the information relevant to a particular directory: file names and records for individual properties. By clustering this data we benefit from locality of reference. Because of our assumptions about the number and size of directory entries, we chose not to use an external (on-disk) representation that would allow manipulating just part of the database; instead, the entire database is read into the server process' address space, converted to a convenient internal form, manipulated, and, if modified, written back to disk. For a directory of moderate size, the cost of reading and writing the whole database once is much less than the cost of repeatedly reading and writing small pieces; this is because a large disk transfer does not cost significantly more than a short one.

A disadvantage of the in-memory approach is that changes made just before a crash may be lost. Rewriting the entire database after every modification would solve this, but would make modifications quite expensive. Therefore, our system instead writes a log of changes that may be used to "scavenge" the database after a crash. The efficiency advantage of logging over complete write-back depends on the availability of fast stable storage, such as a dedicated logging disk.

Support for global queries (that is, queries over an entire directory or file system) requires additional data structures. We use an inverted index for each property name, mapping a property value onto those directories having that value for the given property. Since these indices will be large and modified frequently, we chose to use B-trees as external index structures. Because of the additional storage and run time costs of maintaining these B-trees, one might choose to provide these inverted indices only for a few important properties.

2.6.1. Performance

The storage efficiency of the implementation described above is worse than that of a conventional system such as Unix, because a general-purpose system is inherently less space-efficient than a special purpose system. Our system requires about three times as much storage space for properties as does Unix. (We give detailed performance figures in section 8.3.2.) Compared to the disk space required for the file data itself, this is still minimal: much less, for example, than the space lost to internal fragmentation.

---

[2] The use of this global database resembles the way Lisp symbols are "interned."

The run-time costs associated with our system compare similarly to Unix. The elapsed times reading and writing properties are from 1.7 to 5 times worse than Unix, depending on the nature of the operation. While our system is clearly slower than Unix, it is not painfully so. The property list operations are sufficiently fast to keep up with human interaction — the system can iterate over 650 properties per second, and repeatedly access a single property 1100 times per second.

Our system is several orders of magnitude *faster* than Unix when used for searching for files, over the entire file system, based on property values. This is because Unix provides no support for such searches. The auxiliary indices required for our system to support global searches do add some cost: about 10% per property in both storage space and property-write time. They do not affect property-read time, which is by far the dominant case.

Our model is not meant to compete with the storage and run-time efficiency of existing systems. It is meant to provide capabilities which simply do not exist in conventional systems, and thus to improve the efficiency of the users. Remember, too, that we are only discussing the efficiency of property support; normal data transfer efficiency is unaffected.

2.7. Summary

The point of this thesis is that a coherent approach to file properties, missing from existing file systems, is necessary and often sufficient to properly solve a large class of problems. These problems span the range of interactions with files, but are especially prevalent in heterogeneous distributed systems.

The "proper" solutions that our model enables may not be the only possible solutions, but many of the existing solutions are so cumbersome, incomplete, or inefficient that in practice they are useless. In particular, although we will show how to implement our designs efficiently, the performance improvement does not come primarily from our efficient implementation. Rather, it comes from a conceptual shift that provides the possibility of replacing *ad hoc* solutions with carefully designed ones, linear-time algorithms with logarithmic-time ones, and incompatible interfaces with compatible ones.

In the chapters that follow, we go into far more detail on both the "why" and "how" of a coherent approach. One should keep in mind that this thesis is about better solutions to real problems; real solutions require difficult compromises. The art of computer systems design is choosing the right compromises, and that is what we have tried to do.

Chapter 3
Historical and Related Work

In this chapter we discuss other work relevant to file properties. We begin with a brief discussion of some possible roots of the concept of a file property. We then describe existing systems, or system designs, that include support for file properties. After digressing to examine work on the transmission of structured values, such as properties, in a distributed system, we look at several approaches that have been taken to file properties in computer networks and distributed systems. Finally, we close with a review of several models for file systems and directory systems.

3.1. Historical roots of file properties

It appears that the primary inspiration for extensible file property support comes from the LISP concept of a property list. A *property*, in LISP, is a named value associated with a symbol. In most LISP systems, the properties associated with an atom are collected in a list, called a *property list* [McCa62]. Note that in LISP, the property list is associated with a symbol, not with an actual object; this can cause some confusion when a symbol is bound differently in different scopes, and so LISP property lists have become somewhat unfashionable.

However, property lists are useful for associating information with symbols (such as the symbol's *print-name*, the character string to be used when printing the name of the symbol), and in general for describing objects represented by LISP values. LISP, normally an interpreted language, naturally led to integrated programming environments, in which it was convenient to treat a file as one might treat any other LISP object. For example, the INTERLISP-10 system [Teit75] includes a "file package" that uses and maintains a small set of properties for each of the files that it knows about.

Meanwhile, in the non-LISP world, file systems had been keeping progressively more information about the files they stored. This was not user-defined information; each field in the "file header" or equivalent was defined by the file system designer for a particular purpose. When a new field was required, it could not be added until the next release of the file system. Over the years, a system might acquire quite a few of these attributes; an IBM "Virtual Storage Extended/Virtual Storage Access Method," or VSAM file, might have over 150 attributes [IBM79]. Designers of other file systems chose to maintain a minimal set of attributes, rather than be forced to change the file system interface specification.

One final thread leading to property-based file systems was the concept of *Hypertext.* The term was introduced by Nelson, in 1965 [Nels65], although Nelson cites the visionary 1945 paper by Bush [Bush45] as the original inspiration. Nelson defines hypertext as "a body of written or pictorial material interconnected in such a complex way that it could not conveniently be presented or represented on paper." Hypertext and its successors demonstrated the utility of a connected network of textual data, as opposed (or in addition) to a simple hierarchical file system. Nelson's implementation of Hypertext is described in [Nels81b].

The first Hypertext-like system was NLS (which stands for On-Line System) [Enge68]. NLS maintained a hierarchical network in which the constituent nodes were phrases or larger sections of documents. Dynabook [Xero75], which had a flavor of Hypertext, together with NLS led to the Xerox Alto personal computer [Thac82] and Smalltalk [Gold83]. Smalltalk provides the environment for two more recent Hypertext-like systems: PIE [Gold80], which is essentially a software development system, and Notecards [Xero85a], which has a sophisticated user interface for displaying and manipulating nodes.

These three threads — LISP ideas of extensible data structures applied to files, the strains created by rigid attribute support in non-LISP systems, and the imaginative vision of hypertext — all began to come together in the minds of file system designers during the early 1970s. In the following section, we will see where this led.

3.2. File systems supporting properties

A perception of the need for file property support apparently arose independently at least twice, contributing to three families of file systems that provide at least some support. These were the LISP systems, as noted earlier, and the personal workstation systems developed at Xerox.

3.2.1. Xerox Alto and subsequent file systems

The file system for Xerox Alto workstation [Thac82] may have been the first to provide extensible property support [Xero79]. A file, in the Alto file system, included a *leader page* that conceptually preceded the contents part of the file. The leader page included conventional intrinsic properties, such as write-time, in positional notation, but also had space for arbitrary properties. A property began with two integers, encoding the *type* of the property and its length, in bytes; interpretation of the rest of the property depended on the application. Since there was no map between type codes and property names, the type codes had to be reserved by some administrative mechanism; however, apparently only four such codes were ever reserved.

The Alto mechanism had other limitations: the space available for properties was limited to about 420 bytes; the intrinsic properties were formatted differently from the extensible ones; there does not appear to have been a well-documented package for accessing the properties. The most important flaw, however, was not in the file system itself, but in its environment.

The Alto was designed as a personal computer to be used in a network that included file servers. Thus, the disks attached to the Alto were relatively small (2.5 Mbytes) and were intended for use only as local caches of files normally stored on file servers. Unfortunately, at least some of the file servers could not store the leader-page file properties. Also, although the file transfer protocol (FTP) used to store files on the servers could encode arbitrary properties, the FTP programs written for the Alto did not attempt to transfer the extensible properties. Thus, the only practical use for the extensible properties was with files that were expected to remain on the local disk; this is probably why they were not heavily used.

The Woodstock File System (WFS) [Swin79] had a file property mechanism similar to the Alto's. It was not extensively used.

The Alto and WFS were research projects. Other groups within Xerox were working on commercial products, the Star professional workstation in particular. The original design for the Star file system [Dala86] was tightly integrated with application-level functions. It was then realized that clearer separation of file service and application would yield a more open, flexible

system, but the resulting file system retained some features of Star that were found to be generally useful. One of these features was support for extensible attributes, including attribute-based search.

The resultant *Xerox Filing Protocol* (XFP) [Xero85b] defines a filing facility that may be implemented in various ways. XFP has extensive support for file attributes; it comes closest of any system to providing the facilities suggested in this thesis. XFP attributes are extensible and typed; they are not individually protected, nor are they named. Instead, each attribute is identified by an integer; mapping between attribute numbers and attribute names is done externally to the file system. Because of this early binding, an "uninterpreted" (i.e., non-intrinsic) attribute has no meaning except to those applications that share a map. Also, it may be difficult to add new intrinsic attributes to the XFP specification, since there is no mechanism to prevent attribute number assignments from colliding with existing applications.

XFP provides nicely integrated support for attribute-based search. The **Open** procedure specifies the file to be opened using a small subset of the intrinsic attributes; file name is one such attribute. The **Find** and **List** procedures, respectively, open a single file and list all the files matching a complex predicate on attributes. All three procedures limit their search to a specified directory; XFP does not support global attribute-based searches[1].

3.2.2. LISP systems

At some point between 1975 and 1983, the INTERLISP manual [Xero83] acquired a brief section on "file attributes." In the interim, INTERLISP had evolved from a facility for use on a large timesharing system to become a complete environment for a personal workstation (originally, the Alto). Attribute operations similar to GETPROP and PUTPROP could be applied to any file in the file system, not merely those under the control of the file package. However, unlike the properties maintained by the file package, the file attributes are not extensible.

Meanwhile, along another branch of the LISP family tree, a convention was established for storing property lists in text files. It is possible that this was inspired by the "local variables" mechanism in the EMACS text editor [Stal81], but in any event arose at about the same place and time (the MIT AI Lab in the late 1970s). The personal LISP machines developed at MIT [LISP77] evolved into the Symbolics LISP Machine system [Symb85], which has a convention for storing lists of attributes at the beginning of text files, and some minimal support for manipulating them. This mechanism, unlike the INTERLISP attributes mechanism, is extensible. The Symbolics system also supports extensible, uniformly-accessed, user-defined properties attached to files stored on the local file system; however, since users typically move files between the local file system and remote, non-extensible, file systems, only the embedded attributes are reliably available; existing applications do not use the extensible file properties mechanism. The Symbolics system thus suffers from the same flaw as the Alto did; extensible properties are "trapped" on the local disk. However, unlike the Alto, the Symbolics system uses late binding for property names and thus is far more convenient to extend.

[1] **The search actually covers a specific directory and its descendants to a specified depth. The default depth is 1, meaning no descendants. It is unlikely that XFP intends for a search over a large tree to be efficiently implementable.**

3.3. Transmission of data in heterogeneous distributed systems

In a heterogeneous distributed system, structured data must often be reformatted when it is transferred between hosts. The problem of knowing what and how to translate has been approached in several ways.

Some of the earliest work on this topic was done for the National Software Works (NSW) project [Holl81]. The NSW File Package [Cash76] maintained type information for each file known to it; when a file was transferred from one machine to another, the data could be translated using structural information indicated by the file's type. Another early effort was the Experimental Network Data Manager (XNDM), built at the National Bureau of Standards [Kimb81]. XNDM was an attempt to tie together a heterogeneous collection of database systems; it did so by translating queries into and out of a common representation; it also had to translate data records.

Both NSW and XNDM were concerned with transmission of high-level constructs: files and database operations. Later work has been done mostly in the context of Inter-Process Communication (IPC). Once it was realized that a distributed system could best be constructed based on a uniform IPC, it was clear that in a heterogeneous distributed system the heterogeneity must be faced squarely in the IPC.

The Name-Type-Value (NTV) protocol [Low80] is a machine-independent protocol for the transmission of messages in the PLITS distributed system [Feld79a]. Machine independence comes because NTV messages are self-describing; that is, within a simple framework, it is possible to interpret the typed values sent in the messages without additional information, such as symbol tables. NTV can carry any pointer-free PASCAL data type, including arrays and records, and does not limit the precision or range of integers or reals. Since values in NTV messages are named, NTV could be viewed as a mechanism for transporting property lists.

The Courier Remote Procedure Call protocol [Xero81] definition specifies a representation for transmitting typed data. Although the protocol is intended for use between a variety of hosts and programming languages, the messages are not self-describing; the value of a data object is encoded, but not its type. Type information, and thus interpretation of the values, must come from knowledge shared by the communicating processes.

Herlihy [Herl82] describes a mechanism for storage and transmission of abstract data types between different implementations of a data abstraction. The implementations might be for different machines, or different versions of an implementation for one machine. However, he assumed the use of a single programming language.

The Clearinghouse [Oppe83] is a decentralized database used for locating objects — primarily servers and people — in a distributed system. Entries in the database consist of bindings between names and extensible property lists. Clients can perform various operations on property lists, apparently via remote procedure call; property values are always strings. Clearinghouse seems to be the first distributed system to use the property-list concept for storing and communicating extensible information, and was one of the main inspirations for our approach.

3.4. File properties in distributed systems

In a computer network or distributed system, users naturally want to move or share files between hosts. Early networks provided only a File Transfer Protocol (FTP), which allowed a user to copy a file from one computer to another. Only the Xerox Pup FTP protocol [Bogg80] provides any support for transferring arbitrary properties; many FTPs do not transport any properties at all.

File transfer protocols create a copy of a file; this has the disadvantage that it may require transferring much more data than the user is interested in. It also makes concurrent sharing impossible. An alternative to using an FTP is a File Access Protocol (FAP), providing direct access to parts (pages or arbitrary extents) of a file. Since only one copy of the file exists, sharing is possible. One such FAP is the Leaf protocol [Mogu81], developed at Xerox. Leaf has no mechanism for transferring file properties, except that it provides access to the leader page of Alto files.

This lack of support for transferred properties discouraged attempts to share files within a network, although it is fair to say that the greatest stumbling block to sharing was the lack of transparency in remote file access; programs could not get at remotely stored files in the same way as they accessed local files. Transparency was recognized as desirable, so work was done on building transparent file systems out of loosely-coupled computers. The LOCUS system [Pope81] is the best-known example of a highly-transparent system, but there are many others. In these systems, it should make no difference to a program where a file is stored, and so file properties must be treated transparently.

An unfortunate feature of these transparent systems is that they are in two senses completely "closed": first, a host that is not a member of the system cannot access its files except by using an FTP. Second, and crucial to the problem of property transparency, all member hosts must run essentially identical file systems.

LOCUS maintains the Unix model of the file system as a function of the kernel. More recent systems, following the server-based distributed system model [Wats81], treat the file system as a server process with which client processes communicate using an IPC mechanism. By making the communication explicit, these systems encourage extension to a distributed system: the file system is an independent service, not a function of an operating system [Isra78]. Examples of file servers include the Cambridge File Server (CFS) [Dion80], the Xerox Distributed File System (XDFS) [Mitc82], and the Sun Microsystems Network File System (NFS) [Sand85]. There are also *page servers*, which provide access to disk pages without imposing any file structure; presumably, file structure is imposed in a higher layer. Page servers are thus not of much interest here, since they do not provide anything to which a file property could be attached. One example of a page server is the proprietary Sun Microsystems Network Disk (ND) system [Sun85].

Use of a general purpose IPC for communication between file servers and their clients is an important step toward managing heterogeneity, since by solving the low-level heterogeneity problems in the IPC, it encourages the use of uniform interfaces at the level of the file system. A file server can be designed to be used with a heterogeneous assortment of clients. It is here that uniform access to file properties must be provided, since in a heterogeneous environment one cannot get by with leader pages or idiosyncratic function calls[2]. Several existing file server designs include uniform support for properties, for example the Xerox Alpine file system [Brow84] and the Compact Disk File System (CDFS) [Garf85].

---

[2]Although NFS is supposed to be "independent of Unix" [Sand85], in fact the file property operations are effectively identical to those of Unix [Sun85], and thus are not appropriate for a truly heterogeneous environment.

One other style of loosely-coupled heterogeneous distributed file system is of interest; it has been called the "whole-file-copy" model. Like FTP, it involves creating a complete copy of a file at the client's node, but the client is expected to return a copy to the server if any modifications are made. The server supports sharing by maintaining locks on active files.

One such system was the file handling facility [Cash76] of the National Software Works, an attempt to tie together quite dissimilar systems into a coherent whole. NSW did not survive, partly because the systems it linked were too dissimilar; they lacked a sufficient set of common primitives. NSW avoided the need for remote file access by using FTP to move a copy of a file to the host where it was needed. Because the copy was stored in the file system of the computer where the program that used it ran, transparent access to properties was not an issue. The issue instead was how to copy the properties along with the data, and it seems to have been mostly ignored. Although NSW did keep detailed file type information that allowed structured files to be translated, during the copy process, into a representation suitable for use on the local machine, it appears to have been impossible to determine, for example, the last modification time for the actual file (as opposed to the local copy), or even its owner.

More modern examples of whole-file-copy file systems include Alpine and the Vice file system being built at Carnegie-Mellon University's Information Technology Center [Saty85]. However, both these systems are meant to function in a homogeneous environment; whole-file-copy is used to improve performance, not to support heterogeneity. The Swallow Repository [Svob81] is an object-oriented system that may be viewed as a whole-file-copy file system; since Swallow efficiently supports small objects, it might be a good basis for implementing a property-based system.

3.5. File and directory system models

One of the contributions of this thesis is to propose models for both files and directories that include the concept of properties. These are probably the first models to do so, but others have tried to produce comprehensive models, and so have influenced ours.

We can identify two classes of models. Members of the first are essentially conservative: they describe a "least common denominator" of a large class of existing systems, and thus provide a framework for supporting portable programs. Members of the second class are utopian: they aim to describe an elegant, complete model upon which future designs may be based. The model described in chapter 4 falls into this second category.

Svobodova, in [Svob84], surveys the prominent distributed-system file server designs. Although she focuses more on their implementations than on their external characteristics, she does attempt to classify servers according to some broad concepts.

3.5.1. Conservative models

Conservative models of file systems are not so much explicit models as they are attempts to create a common kernel of operations among a wide variety of systems. Their intent is to support portable programs, or heterogeneous distributed systems.

The National Software Works was an active system that tried to mask the differences between systems, rather than eliminating them. Thus, the file system model presented by the NSW File Package [Cash76] did not have to rely entirely on the facilities provided by the host file system; it could maintain additional databases to augment the information stored by the underlying file system. NSW was a distributed system, and thus required this additional information so that it could automatically translate files when they were transferred between hosts.

In contrast, the Software Tools Virtual Operating System [Hall80] aimed at portability of code, not interoperability, and apparently did not maintain additional databases. The file system model for this system was modeled on Unix, but apparently did not include the Unix file attribute operations.

3.5.2. Utopian models

People who write about Utopian models tend to use the word "universal." One such model is the Universal File Server (UFS) [Birr80], essentially a collection of principles for the design of file service in a distributed system. UFS does not cover file attributes or properties.

A central component of any large distributed system is a directory service, for naming files and other objects. The Universal Directory Service (UDS) [Lant85] is a framework for naming in a heterogeneous distributed system. UDS directory entries, to support a wide variety of name syntaxes, are "sets of *{attribute, value)* pairs": that is, property lists. UDS property lists are extensible, to allow caching of arbitrary hints in the directory entries. The UDS design also supports associative naming, thus allowing objects to be located based on their attribute values rather than their position in a directory graph.

A separate set of efforts have been aimed at developing a model for "mass storage systems," file systems used for storing enormous amounts of data. These are typically used with super-computers doing large numerical analysis problems. Collins and Miller [Coll84] describe such a model; they make vague references to support for file attributes, presumably aiming for uniformity, but do not discuss extensibility.

3.6. Summary

The work described in this thesis is a synthesis of evolving ideas from various fields of computer science. From the evolution of "traditional" file systems, typically those provided with commercial operating systems, we see the necessity for storing large sets of file attributes. From the evolution of distributed systems, especially distributed file systems, we see the necessity for support of heterogeneity. This does not simply mean uniformity or operating system independence; it also requires extensibility.

From the Lisp paradigm we draw the basic concepts of extensible data structures, specifically property lists, and the automatic dynamic storage allocation necessary to make them work. From the work, not otherwise mentioned here, on database systems, especially relational databases, we draw additional techniques for locating data other than by name.

Finally, from the work on Hypertext and related systems, as well as the use of complex "knowledge bases" in artificial intelligence, we are inspired to make use of richer data structures, and data manipulation tools, to improve our use of computer file systems. Not only do we seek more efficient solutions to file system problems within the conventional paradigm derived from paper files, but also better solutions to information management problems within the possibilities unleashed by the computer.

Part II

Structures

Introduction to Part II

We now proceed to describe the various structures upon which we can build solutions to the problems discussed in part I. These structures are of varying levels of abstraction, ranging from a conceptual model, through interface designs, to specific implementations.

First among the structures is a model of files, file properties, file systems, and directory systems, that allows us to discuss solutions to our problems. The model is laid out in chapter 4.

We then digress slightly, in chapter 5, to look at how existing file and directory systems support file properties, and what uses these systems support.

Chapter 6 presents explicit designs for the interfaces of a file system and a directory system, both providing substantial support for file properties. Although the basic design principles are simple, the specification of real system designs requires attention to a swarm of details, resulting from the interaction between the various features of the system. Many of the design decisions are aesthetic by nature, and so these designs represent one point in the space of possible designs.

To show *how* these designs may be implemented efficiently, in chapter 7 we describe several pilot implementations of the interesting parts of the directory system design. Drawing from the failures as well as the successes, we extract a number of principles to guide future implementations.

To show *that* these designs may be implemented efficiently, chapter 8 presents performance measurements of our implementation, and comparisons of its performance against more traditional systems. Because the implementation is meant more to address future applications than to support those that are currently used, in order to compare performance we must make some assumptions about the way these future applications operate; we provide some justification of these assumptions. Finally, we analyze the actual relative performance of our implementation on a few interesting applications.

Chapter 4
A Model of File Properties

Before we can discuss the use and implementation of file properties, we must describe our model of what they are. This chapter begins with a discussion of the environment for which this model is meant. After a brief model of files, we present a model of file properties. We then discuss how properties are associated with files, and the connection between properties and file versions. We conclude with a functional abstraction of operations on properties.

This model is meant to be intuitively reasonable; others are possible. In particular, is not meant to be completely general. Generality as a virtue extends only so far, and then becomes a basis for nit-picking that can distract us from other goals. Thus, the model is not meant to cover every conceivable use of files and their properties, but rather those uses sufficient for most purposes.

There is always a danger to designing a system that is almost, but not quite, a general purpose database: people will complain that it does not do everything "it should." On the contrary, it is a poor idea to design a system that is much more general than its goals require, since its performance in satisfying those goals may suffer. The question of whether a general-purpose database should be used for representing file properties is still open.

It is appropriate to quote from the description of the *Clearinghouse*, a system with similarly limited goals and structure:

> We did not design the clearinghouse to be a general-purpose, relational, distributed database nor a distributed file system, although the functions it provides are superficially similar. It is not clear that network binding agents, relational databases, and file systems should be thought of as manifestations of the same basic object; their implementations may well require different properties. In any case, we certainly did not try to solve the "general database problem," but rather attempted to design a system which is implementable now within the existing technology yet which can evolve as distributed systems evolve [Oppe83].

4.1. Environment Model

Although single-processor computer systems are an important degenerate case, the *'server-based" distributed system [Wats81] is the most appropriate model for our use. In this model a system is made up of a communications kernel and a set of communicating processes. A client process makes requests of a server process to obtain a particular service; a single process may at once be both a client and a server. If the communications mechanism is adequate, client and server may be on different machines. For example, file service may be provided by a server process on a machine with a disk, but accessed from a client process on a diskless workstation.

This differs from older models known as *monolithic* operating systems, in which the file system is an integral part of the kernel. In some cases, for example Hydra [Wulf81], only the lowest level operations are in the kernel; those operations which can safely be performed by

user processes are done in library routines. Although monolithic systems can bring certain performance benefits, the price is a proliferation of system calls to support file service, high resistance to experimental file system designs, and, most important, difficulty in creating distributed systems. It is not that difficult to add file server processes to an existing monolithic system, but "client" programs of monolithic systems are often not able to run in a distributed system without extensive modification.

There have been numerous attempts to tie multiple monolithic systems together into a distributed system. LOCUS [Pope81] is a notable example; other Unix-based approaches include the Newcastle Connection [Brow82], and Datakit [Fras79]. In the commercial world, Digital Equipment Corporation's VAXclusters system [Kron85] and Systems Industries' SIMACS [Chu84] have been successful.

By providing significant location transparency and a unified name space, such systems make it possible to use unmodified client code. Since they are based on monolithic systems, they can avoid a lot of overhead in process switching and other layer-crossing operations, and so are potentially quite fast; LOCUS has demonstrated performance on remote operations nearly indistinguishable from that on local operations. However, these designs almost always support only component systems of identical structure. Because they are so homogeneous, and "closed" (in the sense that only a predefined set of systems can communicate), from the file system point of view it is perhaps correct to treat them not as distributed systems, but rather as loosely-coupled multi-processor systems.

While the homogeneous distributed system is useful as far as it goes, it leaves a big problem unsolved. Many organizations have a variety of computer systems out of which they would like to create a distributed system; this will probably always be the case, even as older systems are replaced with newer ones. So, we would like our solutions to work in a *heterogeneous* environment: a distributed system made up of a variety of differing component systems, and with indistinct boundaries.

4.1.1. Separation between Directory and File Systems

The old monolithic system model usually implied a tight coupling between file system and directory system; in many cases the two are indistinguishable. While this approach might improve performance slightly, one of the recognized virtues of the client-server model is that it encourages separation of function. Separating file system and directory system into two distinct services (which might nevertheless interact as clients of each other) provides several benefits:

- **Modularity**: with attendant benefits of a cleaner service model, cleaner implementations, and the flexibility to substitute new implementations of one service without affecting the other.

  Modularity has additional value: it is easier to distribute specialized services than to build a generalized distributed system. For example, LOCUS uses its knowledge of the special semantics of directories to recover them after a partition; this cannot be done for files in general [Pope81].

- **Crossing file system boundaries**: an integrated directory system can only store references to files in its associated file system. In a heterogeneous distributed system, we would like to construct a unified name space covering all file servers in the environment. Why should we prohibit users from storing in the same directory references to files stored by two distinct file servers? The Universal Directory System [Lant85] is an example of an approach to this question.

- **Non-file referents**: Directory systems can and should be used as general name-binding agents; they need not only refer to files. In fact, directories have been used to name non-file objects even in Unix (where devices appear in the file system name space). By separating directory from file service and giving non-file referents full citizenship in directory bindings, we can gain useful generality.

Not everyone agrees that directory service and file service should be separate; for example, the V-system takes the opposite point of view [Cher84]. We prefer separation because it leads to a simpler model.

4.1.2. Communications Model

In a traditional, single-CPU operating system, communicating between the "file server" and its clients is by system call. If some of the "clients" are actually other modules within the kernel, they may communicate with the file system using subroutine calls, interrupts, queues, etc. All of these mechanisms obscure the process of communication, by blurring the identities of the communicants.

A clean Inter-Process Communication (IPC) mechanism, which makes communication explicit and uniform, is a fundamental facility for a modern operating system. Unlike systems that rely on implicit mechanisms and service-specific system calls, IPC-based systems allow clients and servers to be separated across a network without changing the basic client-server structure. Because it is almost impossible to build a distributed system without a good IPC, we will take it for granted as part of the environment.

The easiest IPC to implement is an unreliable synchronous message system, or datagram protocol. It would be nice if we could create service-level protocols that could make use of datagram protocols, but use of unreliable protocols requires that programmers re-implement reliability mechanisms at high levels that might have been provided in the IPC layer[1]. Saltzer et al. [Salt84] argue that this re-implementation must be done in any case, since one cannot detect high-level errors in a low-level protocol. However, the programmer's life is made far easier if the IPC mechanism meets reasonably high reliability goals. A reliable, sequenced, synchronous message protocol is an appropriate compromise.

One such protocol is Remote Procedure Call (RPC). RPC is an attempt to preserve the semantics of procedure call for communication between processes. Its value lies in its familiarity to programmers. Because of this, we can use it as a notational convenience in describing operations on files and file properties. However, this does not mean that file property operations must be done using RPC; any number of other IPC mechanisms could be used. For a thorough discussion of RPC, see Nelson [Nels81a].

---

[1]One could use a non-duplicate-suppressed protocol if the service-level protocols are idempotent.

4.2. What is a file?

> *[Operating systems provide] powerful, hierarchical abstraction principles that permit users to operate on idealized versions of resources without concern for physical detail —for example, processes instead of processors, files instead of disks, data streams instead of program input/output.*
>
> — Peter Denning [Denn85]

> *Plastic is not imitation anything, it is real plastic.*
>
> — Peter Max (attributed)

Files are not imitation disks, they are real files. Before we can discuss file properties, it would be nice to have a clear idea of what a real file is. Such a definition cannot be rigorously formal, since we are interested in a heterogeneous, open environment, and so we should not be too restrictive. On the other hand, one cannot design a practical system to handle files without ruling out some of the wilder possible definitions.

> **Definition;** *a* file *is a named object that stores an arbitrary amount of data for an arbitrarily long time.*

The key phrases in this definition can be analyzed, to see how files differ from other objects:

- Named object: In some sense, every object in a computer system has a name. However, it is important that a file's name be meaningful in some global context, for otherwise one could not share it very widely.
- **Arbitrary amount of data:** Unlike program variables, which are usually of fixed size, a file can store any amount of data; the size of a file need not be predetermined.
- **Arbitrarily long time:** Unlike program variables, which disappear when a program terminates, or when a computer crashes, files are expected to persist; they are ''stable storage."

One should not confuse these definitional phrases with goals for specific implementations. The maximum size of a file must be an implementation parameter; it is not reasonable to require the storage of indefinitely large files. Similarly, stability of file storage is a reliability goal; no file system can guarantee absolute reliability, so one must always accept some risk of losing a file.

4.2.1. Mutability and Deletion

Two aspects of many file systems that are not part of the definition given above were omitted because they are not universal. One is the ability to change a file once it has been created; the other is the ability to delete a file. Neither of these are fundamental; both may profitably be left out of a file system design. For example, neither Hydra [Wulf81] nor Unix [Ritc78] has an operation to delete a file; the Cedar file system [Schr85] does not allow modification of an old file version.

What does it mean to change a file? In a file system based on magnetic storage, the abstract

operation of changing a byte in a file might cause the physical operation of simply rewriting one disk block in place. This might be called the ''Von Neumann'' paradigm of files.

However, it is also possible to treat files as immutable values; each change creates a new value, but the old value is retained as well. This might be called the "dataflow" paradigm of files. It is not as strange as it might seem; if we look for an analogy to the world of paper files, we see that one often updates a file by adding or replacing a sheet of paper, rather than using an eraser to remove old writing before replacing it with new information. In fact, it is sometimes illegal to remove information from a paper file.

Immutable-file systems such as Cedar are inherently multiple-version systems (see section 4.2.2), because they retain old file values. There is a conceptual clarity gained by immutable files, since sharing is simpler to understand. Immutable files are useful for storing text, especially program text, because source code version management depends on accurate records of changes. Immutable files are also a good match for write-once storage, such as optical disks.

There are cases where the use of immutable files is awkward and inefficient. For example, representing a large on-line database as a sequence of versions of an immutable file is possible, but the copy overhead is prohibitive[2].

Deletion of files is another file system feature that is not actually fundamental. With write-once storage media, actual physical deletion no longer serves its current primary purpose of releasing storage space for other use. As a policy issue, sometimes deletion should not be allowed; destroying evidence of a crime, for example, is much harder if files cannot be deleted. On the other hand, there are some circumstances (such as records of juvenile offenses) where files must be deletable after a certain period. It would be necessary to physically destroy write-once storage media to carry out this policy.

4.2.2. Versions

With or without immutable files, there are times when users want to retain old versions of files as well as current ones. For example, old versions might be retained merely for historical value, or as backups in case a change must be rolled back, or to maintain multiple versions of the same program. Some file systems explicitly support multiple versions; others do not. True support for multiple versions requires more than an extended name space; it is intimately related to support for file properties, as we will see in section 4.4. Explicit version support is desirable, but not required, in conjunction with support for file properties.

Whether or not multiple versions are explicitly supported, the model used in this thesis treats a file as a history, or sequence, of versions. We use the term * 'file history" when it is important to refer to the sequence as a whole, and the term "file version" when it is important to refer to a single version. The determination of what constitutes a distinct version is a policy issue; intuitively, one would define a version as a file value that might be referred to in the ftiture, even if subsequent values have been created. Individual states of a database during a transaction should not be considered "versions."

To aid our intuition, consider an analogy with the world of publishing. The revised edition of *Mastering The Art of French Cooking* [Chil83] is clearly distinguishable from the previous

[2] If the structure of the file is well-known to the storage system, it is possible to use copy-on-write strategies to preserve the illusion of immutable files without copying any more than is necessary; for example, the Swallow Object Repository [Reed81] is a design meant to support active databases on write-once storage.

edition, and the distinction is a valuable one[3]. On the other hand, uncorrected galley proofs of this edition, while perhaps distinguishable from the final result, are of no interest to anyone but souvenir hunters, and so are not identified as an "edition."

The importance, for this thesis, of the two concepts "file history" and "file version" is that one can associate certain properties with each. That is, a specific version of a file might have properties that no other version has, but there might also be properties that the file possesses as a continuous entity, over time. Section 4.4 develops this point.

The file history, not the file version, is the basic identified object. If one assigns unique IDs (UIDs) to each file version, then the relationship between a file history and its versions becomes muddied. For example, how would one get at properties of the file history knowing only the UID for an version? Rather, we assign a UID to the file history as a whole, and use a *{UID, Version-ID)* tuple to identify a particular version. This means that a directory entry remains valid even if a new version is created.

4.2.3. File contents

Our definition of a file as an "object that stores data" doesn't say anything about what the data looks like, or how it is stored and retrieved. In designing a practical system for a heterogeneous environment, this must be carefully specified. However, the specification must not be restrictive; it should allow complete portability of any data.

One point of view is that files can have a variety of specific internal structures, which should be understood and supported by the file system. The opposite point of view, more appropriate for an open system, is that any complex structure is provided at higher levels; the file system directly supports only one simple structure. The structure most appropriate for use in a distributed system is an array of bytes; while the choice of a particular byte size is inevitably arbitrary, it is usually convenient to make it eight bits.

For a file treated as an array of bytes, there are two basic ways to access it: as a sequential stream, or as a random-access array. The random-access model is far more efficient when dealing with inherently unordered access patterns, such as in a database application. Any other access method, such as sequential, can be mapped onto random access at a higher level; the converse can not be done efficiently[4]. Also, random-access operations in a distributed system can be cast in a form which provides, if not idempotency, at least the ability to retry an operation repeatedly in the face of communications failures.

By using the byte as the basic structure, we avoid any incompatibility due to incommensurate page sizes. Therefore, in this thesis the random-access array-of-bytes model will be used, as it is quite general and thus consistent with a heterogeneous distributed system.

---

^e first edition did not make use of the food processor.

[4]While efficient sequential access requires particular buffering and pre-fetehing assumptions, this does not mean that a sequential access layer implemented on top of a random-access file model must be inefficient Intelligent buffering can still be done, either in the sequential access layer, or even in the random-access file system: the file system can detect sequential access patterns, or be advised by a higher layer to use a different strategy. Implementing a random access layer over a sequential-access file system is *inherently* inefficient, since non-sequential access then takes O(A0 instead of O(log(A0) time, where *N* is the length of a file. Thus, a random access *mechanism* is more general than sequential access *mechanism,* and it admits a sequential access *policy.*

4.3. What is a file property?

In chapter 1 we used an intuitive definition of a file property; we can now introduce a more formal one.

> **Definition**: *a* property *is a named, atomically accessible, typed value associated with an object. A* property name *is the name of a property associated with one or more objects. A* file property *is a property associated with a file object.*

In other terms, a property is the mapping (*Object-ID*, *Property-Name*) ⇒ (*type*, *value*).

Let us analyze the key phrases in the definition:

- **Named**: Every property has a name. For a given object, there is at most one value associated with a property name. There may be no such value associated, in which case the property is not considered to exist. There can be no more than one value for a property at a given time, although the value may change over time.

  Property names should be human-sensible: they should be character strings such as "Length", "Protection", etc. Names should not be case-sensitive, since having different properties with the same spelling would lead to confusion.

- **Atomically accessible**: A property value can be read or written only as an atomic whole. It is not possible to change part of a value, e.g., a few bytes in a character string, except by changing the whole value. A client can do a partial update using a read-modify-write transaction, but the file system must perform any single-property update or read atomically.

- **Typed value**: If a property exists, it has a value. Property values may be of a limited variety of types (see section 4.3.1). A property value is "compact"; if one wishes to store a lot of information, a file should be used, not a property value.

- **Associated with an object**: A property is not an object; it cannot exist in isolation. A given property is associated with one object. Two objects may have identical properties, but these are copies, not shared.

Although it might seem strange, we do not require that two properties with the same name (associated with different objects) be of identical type. This is because we do not insist that two properties with the same name, even with identical values, have the same meaning. The association between property name and meaning is imposed at a higher level of abstraction; this is discussed further in section 11.1.

We augment this definition by associating protection information with each property, so that a property can be protected independently of the object with which it is associated. With protection included, the mapping becomes (*Object-ID*, *Property-Name*) ⇒ (*type*, *value*, *protection*). The meaning and representation of property protections are described in section 4.3.2.

4.3.1. Property Value Types

We followed these criteria in choosing the data types provided for property values:

- **Diversity**: We need more than one property type.

- **Economy**: The set of property types should be small.

- **Compactness:** Values should have compact representations.
- **Implementability:** A type must be efficiently implementable.

Why do we need more than one property type? We could allow only one type, such as character string, and use it to encode other types. However, encoding everything as a character string is an inefficient use of storage space. Worse, it precludes meaningful manipulation of properties by the file system. For example, is *'March 31" less than "April 1"? It depends on whether the system understands that the values represent dates, not arbitrary strings. Since our file system understands values of several types, they must be explicitly representable.

Moreover, if a type structure is not available at the lowest level, then it must be imposed at a higher level. For example, does the string "faux" encode a value of type 'string', or is it an encoding of a boolean value? A type structure at the lowest level provides a convenient, efficient mechanism for recording type information.

We need a variety of types, but not too many. It is unreasonable to provide the types 'English-string', 'German-string', 'Frisian-string', etc., when 'string' would serve in almost all cases. The set of possible types should be carefully limited to those that are frequently necessary. A small set of types implies a compact encoding for type tags, which improves storage efficiency.

Property values are not meant to record copious amounts of data; they are meant to record simple facts. Thus, such types as arrays, linked lists, trees, and other structures are not appropriate for property values. When it is necessary to store a large datum, the "value" can be stored in a separate file, and the property used as a pointer to this file. Compactness means not that artificial limits should be placed on the size of integers or strings, but that these sizes are subject to the usual constraints of our finite computers. An integer should be able to represent the size of the largest possible file, but infinite-precision arithmetic is overkill.

The economy and compactness arguments given above are really implementability arguments. A property type that cannot be implemented efficiently, or cannot be implemented without adversely affecting other features of the system, should not be provided.

Based on these criteria, on analogy to familiar programming languages, and on examination of how properties are used in existing file systems, we can reasonably expect a system to support the following set of property value types:

Integer Used for common values such as file length, byte size, unique identifier, etc.

Real Potentially useful and not difficult to implement.

Boolean Used for flags such as lock state, "temporary", etc. and useful as described in section 11.5.1.3 to represent relationships between files.

Time Used for creation, modification, access, expiration dates, etc.

Protection Used for access control. However, as discussed in section 4.3.2, choosing a "universal" formalism for protection is problematic.

There are four related variants of the character string type:

String Variable-length character strings are useful as an "escape" in that they can encode virtually any type. A string value is used when pattern-matching or lexical-order queries might be applied.

Case-sensitive **string** String comparisons should by default be case-insensitive, but for some applications case-sensitivity might be required.

Atom Atoms are like strings except that they do not permit any but equality

queries (i.e., no lexical ordering or pattern-matching) and so can be stored and handled far more efficiently. Obvious uses are user names, file types, etc.

Case-sensitive atom

Similar to atoms but case-sensitive. For example, references to Unix files, since the Unix directory system is case-sensitive.

The "case-sensitive string" is the most general, but the most expensive as well. The "atom" is the least expensive type, because it is least general, but it serves the dominant uses.

4.3.1.1. Ordered and unordered types

We will see later, in section 4.6, that some of the operations one might perform on a property value involve inequality comparisons, such as "less than." Such comparisons require a total ordering on the values. There is an obvious ordering for values of type 'integer', 'real', 'time', etc. However, no obvious meaningful total order exists on 'protection' values: while a partial order certainly exists, one cannot always decide if one protection value is "between" two others.

Character strings can be ordered if one assumes a standard collating sequence, such as ASCII. That they can be ordered does not, however, imply that they must be ordered. In fact, one can achieve significant efficiency gains by failing to support inequality comparisons on strings; this is the reason for providing the 'atom' data type, which is unordered, in addition to the 'string' type, which is ordered.

Strings are a special kind of ordered type, because one can also apply pattern-matching queries over such values. There are no obvious applications for lexically-ordered strings; pattern-matching might be useful for partially-specified keywords. Queries of this kind often involve fully-specified prefixes[5], and an underlying lexically-ordered database makes it easier to answer them.

4.3.1.2. Representation choices

Choices of common representations for data types in a heterogeneous environment inevitably clash with some existing systems. The trick is to minimize the need for type conversion without compromising the future utility of the system. In section 6.5, we present a set of representations appropriate for a practical system design. For the purposes of the model, it is important to understand the kinds of compromises that must be made.

A common representation must be capable of representing the values one is likely to encounter, with precision adequate for most applications, but should not sacrifice storage and communication efficiency just to represent an enormous set of values. Conversion between the common representation and host representations should be inexpensive and reversible.

For example, 32-bit two's-complement integers provide a sufficient range as long as we don't expect individual files to exceed two gigabytes in length. Only a few systems are limited to binary-coded-decimal or one's-complement representations, and these are easily convertible to and from two's-complement. Similarly, strings could be limited to several thousand characters without really limiting their applicability.

---

[5]For example, the SOCRATES on-line library catalog [Stan84] supports queries based on the prefix of a keyword, but not general patterns.

4.3.2. Property Protection

As mentioned earlier, we associate protection information with each property. This is distinct from a "Protection" property specifying access to the object itself; it is convenient to allow different access rights to the properties of the file than to the file itself; it is also convenient to allow different rights to different properties.

For example, one might want to allow a friend to write the contents of a file, but not to modify the protection of the file and thus pass on write-access to another user. The first user could also be allowed to modify some properties (other than "Protection"), such as one recording a dependency on another file.

Representing protection values, both for protecting files and for protecting properties, is difficult because there is no simple model that matches all existing protection schemes. This means that we cannot choose a common representation that encodes every possible value; conversions therefore may not be reversible. In section 6.3.2, we describe a restricted scheme that serves the purposes of this thesis.

4.3.3. Intrinsic properties

We add one additional definition:

> **Definition**: *an* intrinsic *property is one that is understood by the server that manages the object with which the property is associated.*

The key word in this definition is "understood." A server understands a property if it either uses or sets the property without direct instruction from a client. For example, a file server understands the "Length" and "Modify-Time" properties of a file, because it sets these as side-effects of write operations. A file server also understands the "Protection" property, even though the property value is completely specified by the client. A "Text-character-set" property, on the other hand, may not be intrinsic, if the file server always treats the file as an array of bytes.

4.4. The connection between properties and versions

Support for multiple version files requires more than just a extension to file naming syntax. Two versions of a single file are related in important ways; in particular, they have identical values for many of their properties. A file system should understand and support this relationship between the properties of versions.

The relationship between versions of a file is an instance of a broader kind of relationship, which we call the *derivation* relationship. A file is *derived* from a set of files (called the *source set*) when it is created by some transformation involving these files as input. Examples of derivations, besides the version derivation, are: copies of files; object files compiled from program sources; and memory dumps resulting from the use of program binaries. A derivation transformation must often assign properties of the source set to the derived file. This requires support from the file system.

For example: suppose you have a file listing the salaries of the programmers in your division, and you have sensibly set the "Protection" property to prevent other people from reading this potentially sensitive information. You run the file through a document formatter to produce

a file to be sent to a laser printer. The formatter creates the output file with your default protection, which allows everyone to read it; the sensitive information is printed that evening by a nosy subordinate.

You would like the output file, derived from the carefully protected input file, to automatically be at least as well protected. There may be other properties that should be preserved across a derivation. The problem is to determine which properties should be preserved, and what agent should do the preservation: the file system or the application.

We take the approach that the file system should provide a *mechanism* for preserving properties across a derivation, while leaving the *policy* decision of what to preserve to higher levels. The mechanism we propose is passive: the file system does nothing unless explicitly commanded. This mechanism could therefore be implemented in the file server itself, or in a higher layer, potentially resident on client hosts. In the design presented in chapter 6, we choose to place this mechanism in the file server, to improve efficiency and especially to provide uniformity: an identical mechanism is available to all clients, because it is implemented only once.

We propose two separate mechanisms: one for supporting the version relationship, and one for supporting more general derivations from single-element source sets. Derivations from multiple files (for example, the compilation of several source files into one object module) are not discussed further, and will have to wait for future work.

4.4.1. Inheritance of properties by file versions

When a new version of an existing file is created, it should have some of the properties of the old version. For example, the "Owner", "Protection", and "Type" properties of a file are usually constant from version to version. On the other hand, properties such as "Modify_Time" clearly should not be preserved.

We use the term *inheritance* to describe preservation of properties from version to version of a file. A property acquired from a previous version is said to be *inherited.*

Some existing systems that support multiple file versions (e.g., TOPS-20) provide inheritance, by copying the "appropriate" properties from the previous version to the new version. This works well for file systems without extensible properties, since the properties are all intrinsic and the system "knows" which should be inherited. A system supporting extensible property lists apparently must face the problem of automatically distinguishing between heritable and non-heritable properties, in order to support automatic inheritance of arbitrary properties. Actually, this is a confusion between policy and mechanism that is provoked by the model of inheritance implicitly followed by existing systems; we propose a different model that avoids this confusion.

We classify the "traditional" model as a *temporal* inheritance model. Temporal inheritance means that a version acquires, at the moment of creation, a subset of the properties of the previous version. Versions of a file do not actually share the inherited properties, but rather have individual copies of these properties. The problem with temporal inheritance is that when one wants to update a property that is logically shared, one must update all the copies. Also, the decision whether or not to inherit a property must be made at the time of version creation. Temporal inheritance suffers from the disadvantages of early binding.

Our model is a *hierarchical* model, more akin to inheritance in such systems as SmallTalk [Gold83] and KRL [Bobr76, Wino75]. The hierarchy referred to is a two-level one, with a file history as the superior node, and the file's versions as the inferior nodes.

Hierarchical inheritance means that all versions of a given file history share the properties associated with the history as a whole. Because an inherited property is shared, rather than copied, it preserves the connection between the versions. For example, one can change the protection of the file as a whole, instead of having to change the protection of each of the versions.

Inheritance is not always the desired behavior; the hierarchical model allows us to override inheritance for specific properties of specific files. If we want one version of a file to be protected differently from the others, the system should handle that special case without disrupting the sharing between the other versions. If we decide later that this version should no longer be protected differently, we should be able to reinstate the inheritance. Further, a client of the file system should be able to distinguish between an inherited and an explicitly assigned property.

Thus, what we want is the flexibility that comes from late binding: the ability to revoke or modify the decision to inherit a property. By providing such flexibility, the mechanism implied by our hierarchical model can be cleanly separated from client policies.

The hierarchical inheritance model is based on the explicit distinction, made in section 4.2.2, between a file history and a file version. Properties that should be inherited by the versions of a file are associated with the file history. Properties that are specific to a version are associated with that version, not the history as a whole. Inheritance of a property by a specific version can be overridden by associating a value of that property with the version, in addition to whatever value is associated with the file history. One can "un-override" inheritance simply by removing the version-specific binding of the property.

If inheritance is wanted, the appropriate way to read a property of a version is to first look for it in association with the version, and if it is not found there, to then look for it in association with the file history. If inheritance is not desired, one only looks for a version-specific binding of the property. The file system design in chapter 6 provides these two different lookup mechanisms, allowing the client to make the policy decision at the latest possible time.

Note that temporal inheritance is available as a special case of hierarchical inheritance. The early binding of temporal inheritance can be implemented as a policy wherein hierarchical inheritance is "overridden" for every extant property of the previous version, and the non-inheriting lookup mechanism is used. Since hierarchical inheritance cannot be provided as a special case of temporal inheritance, it is the more general model.

4.4.2. Preservation of properties across derivations

We now return to the problem of preserving properties across derivations in general. This is much harder than the inheritance problem: while a new version of an existing file is probably sufficiently similar to older versions that it can share most properties, a derived file is potentially of a completely different type from the original file. Many properties therefore cannot be blindly copied.

The combination of inheritance with derivation complicates matters by introducing potential conflicts. If a derived file is also a new version of an extant file, for some properties we must choose between derivation (from the "source" file) and inheritance (from the extant "destination" file). For example: if we run `salaries.mss` through SCRIBE to create a new version of an extant `salaries.doc`, does the new `salaries.doc` version inherit the "Protection" property from the previous versions, or does it get the protection of the `salaries.mss` source file?

In most cases, the inheritance/derivation conflict will not arise. If the earlier versions of the destination file were all derived by the same transformation from earlier versions of the source file, then this conflict will only appear if inheritance is overridden for a derived property. When a conflict is detected, the only safe action for the file system is to complain, rather than to try to resolve without complete understanding of the situation.

In section 6.7.4.4, we specify two operations which together provide support for derivations in most, but not all, cases. The first allows an application to create a file history with properties identical to some other file history. The intent is that the application then modifies those properties that should differ from the source file; all versions of the destination file inherit the remaining derived properties.

The second operation creates a new version with properties identical to those of a version of some other file. This operation must detect conflicts, by examining the properties of the two file histories and the source version. A conflict is deemed to arise if any property associated with the source version would override a property from the destination file history. If a conflict is detected, the operation fails with a distinct error indication.

Unfortunately, no static model of derivation can cover all applications. Our solution is meant to automate the common cases, and to allow for client intervention when necessary. Once we have some experience with this system, we may decide to modify this approach.

4.5. Where are file properties stored?

So far, we have left unspecified the way that properties are associated with objects such as files. We must now define this association, since it is central to the design and implementation of a system based on properties. In this discussion we are concerned with the logical storage "location" of properties, not the physical location. What is important is not on which disk, or which host, a property is stored, but by which server and by what "name."

We will first look at "property lists," an abstraction for collecting a group of related properties. Property lists are a basic organizing principle. We will then discuss "property list groups" a second-level abstract structure that is useful in designing practical systems. Finally, we will look at how property list groups can be used to represent both multi-version files and directory nodes.

4.5.1. Property lists

A *property list* is simply a set of properties[6]. A property list may contain any number of properties, or none at all. A given property name may not appear more than once on a property list, even if bound to different values.

[6]The term "Property List" is somewhat misleading, since the properties are not in any particular order, but it persists for historical reasons, having first arisen in Lisp. "Property set" would be more accurate.

4.5.1.1. Notation

It is convenient to have a notation for property lists, both for showing examples on paper and for constructing crude user interfaces. In honor of Lisp, we can use parenthesized lists of the form

(Property Property .•.)

where properties in turn are represented as parenthesized tuples

(name type value protection)

So, for example, we might have

((Length integer 45 "O:RW") (Type atom "Text" "O:RW"))

Since for the examples in this thesis, property protection will usually be irrelevant, and the property type might be obvious from the value, we will use an abbreviated notation when possible. The previous example could be written

((Length 45) (Type "Text"))

The ordering imposed by this notation is purely arbitrary.

Values of string, atom, time, and protection types will be enclosed in quotes in this notation. This is purely to avoid ambiguities arising from embedded spaces, not to imply that these are all represented as character strings.

4.5.2. Property list groups

Property lists often come in groups. For example, on a file system that supports multiple versions, the property lists for the versions of a single file are logically related. A *property list group* is a set of named property lists.

The value of the property list group abstraction is that it provides a name space for property lists. If several property lists are associated with a single identified object, it is necessary to distinguish them in some way. We have a hierarchical name space for properties: a property is identified by an object ID, a property list name, and a property name.

Property lists and property list groups are isomorphic, in that each is a set of named, lower-level structures. It is tempting to generalize, and replace these concepts with a recursive "list of named (list or atom)" model; this would be a mistake. By limiting the hierarchy to two levels we avoid unnecessary complexity in both our model and in our designs and implementations. We prefer the term "property list group" because it is less cumbersome than "list of property lists," and because we can use the abbreviations "list" and "group" when they are unambiguous.

In the following two sections, we will see how property list groups provide a natural representation for multiple-version files, and for directory nodes.

4.5.3. File properties

The obvious approach to take to file properties is to associate a single property list with each file. In a file system that does not support multiple versions, this one-to-one mapping is straightforward. In a system that does support versions, properties may be associated either with specific versions of a file, or with the file history as a whole. Thus, a one-to-one mapping between file and property list is not exactly what we want.

Instead, we associate a property list with each file version, and one more with the file history as a whole. The set of property lists for a file can be represented as a property list group; the property list names of the version lists correspond to "version numbers." A distinguished name denotes the file history property list.

With the property list group as a representation for properties of multiple-version files, we can make the inheritance model described in section 4.4 more concrete. When a client tries to read a property of a file version that does not appear on the appropriate version property list, the file system can also look for a property of the same name appearing on the file history property list. Thus, to cause a property to be inherited hierarchically by all the versions of a file, it suffices to put it on the file history property list. To override such an inherited property, on a version-by-version basis, one need only to add a property of that name to the appropriate version list.

4.5.4. Directories and directory entries

What is a directory system? It is a system for mapping meaningful names onto low-level names. We can refine this simple view to get a detailed, yet generally applicable, model of directory systems.

We begin by realizing that a flat name space is undesirable. All but the most limited directory systems support a set of name spaces, called *directory nodes* or *directories*. Within each of these name spaces, at most one "entry" is bound to each primitive name; however, a primitive name may be used in many different directories.

Segmentation of the name space provides contexts in which names are interpreted; by allowing the same name to appear in several contexts the system supports multiple uses of a name. This simplifies name management in a multi-user environment. It also makes possible multiple views of a name or collection of names. For example, several implementations of a program may exist, using the same module structure and module names; storing the file names for each implementation in a separate directory is a practical way to avoid confusion.

Name space segmentation also supports logical grouping of named objects. Even a relatively small file system may contain tens of thousands of files; finding a single file in a flat name space of that size would be looking for a needle in a haystack. Instead, we use a set of manageably-sized directories, each storing logically related bindings. For example, it is common to group the files belonging to each individual user in a separate directory, or to group the source modules for a program. The logical grouping function provides name contexts that are small enough for a human user to comprehend, indicates meaningful associations of files, and makes finding a file much simpler. Additionally, by providing a significant locality of reference, logical grouping makes efficient directory searches much easier to implement.

4.5.4.1. Hierarchical directory systems

Use of directories for logically grouping significantly reduces the problems associated with large, flat, name spaces, but directories themselves must be named. Flat name spaces for directories are not necessarily unmanageable in small systems, but in multi-user systems a flat name space is cumbersome. Some systems (for example, TOPS-10, WAITS, RSX-11, RSTS) support a two-dimensional directory name space, usually so that each user can have several directories for different purposes. However, an obvious improvement is to apply the logical grouping paradigm recursively to the directories themselves, yielding a hierarchical directory structure as in Multics, Unix, or VAX/VMS.

A hierarchical directory structure means that directory entries might refer to directories instead of to files. This can be done either by representing directories as files, or by allowing directory entries to refer to various types of objects. There is no reason why only files should be permitted as referents of directory entries; there are many other types of objects that one might want to name in a computer system, and it makes sense to use a single naming system for many object types.

Some systems enforce a strict hierarchy on directory graphs; especially, that directory graphs must be acyclic. Cycles can cause problems for naively recursive graph-walking programs, and can cause confusion by creating aliases. However, neither of these problems is severe, and in a distributed system (where several directory servers may be involved) cycle detection is impractical. At best, one can expect clients to follow a convention, such as the use of "parent" entries in directories, that imposes a spanning tree on the graph.

4.5.4.2. Directory entries

What is a directory entry? A simple formal model is that it is a mapping: (*Directory-ID*, *Entry-Name*) $\Rightarrow$ (*Entry-Value*). In this section, we will look at what the *Entry-Value* really is.

In simple directory systems, especially those where only files can be named, the entry value is usually a descriptor (e.g., RT-11) or reference to a descriptor (e.g., Unix) for a file. The directory lookup operation is implicit in the process of opening a file.

In a distributed system, however, where directory and file system are divorced and the type of the referent cannot be taken for granted, the property list is a better model for a directory entry value. This is because the value is probably complex, and might have different structure for different referents. By using a property list representation instead of an idiosyncratic data structure, we obtain uniformity that is useful in a heterogeneous distributed system.

What properties might be found in a directory entry? Three are more or less required of a directory service:

Entry type — Discriminates between entries that refer to other directories, and other types of referents (files, etc.). This allows interpretation of hierarchical names without requiring constant access to the properties of the objects identified.

Server identifier — Denotes the server that manages the named object; this might a file server, or another directory server.

Low-level name — An identifier for the named object that is meaningful to the server that manages it.

Other properties are useful for a more sophisticated directory service: entry protection, entry creation, modification time stamps, etc.

We make a distinction between *object properties*, properties of the object itself (such as "Length") and *entry properties*, properties of an entry that refers to the object (such as "Low-level-name"). We might use an "Entry-protection" entry property to control access to an entry, distinct from the object property that controls access to the object that entry refers to.

Normally, there is no overlap between entry properties and object properties. However, since one of the common purposes of a directory search is to find a file that meets some criteria, one might "cache" copies of object properties in the entries that refer to them. This can greatly speed such searches, since it is not necessary to examine each object's properties directly. On the other hand, the difficulty of keeping this cache up to date may overwhelm any improvement in search speed. Fortunately, there is a useful class of applications where cache maintenance is not an issue, because the value of a cached property will not change once it is set.

Our model allows the possibility of mechanisms to maintain cache consistency. Because we see no economical way to guarantee consistency in a real system, we chose not to provide these in the design of chapter 6. In chapter 9, we describe some actual applications where cache consistency is not an issue, and caching provides dramatic performance benefits.

4.5.5. Representation of Directories

By representing a directory entry as a property list, we can model a directory as a named set of property lists. This is precisely what we have defined as a property list group; a directory is an object with no value but with an associated set of property lists. The property list group abstraction is useful not only for representing multiple-version files, but also for representing sophisticated directory structures.

4.6. Functional Abstraction

In section 4.3 we started to define a data abstraction for file properties. In this section, we complete the definition by describing in some detail the operations that can be applied to properties. At no point do we wish to expose the physical representation of properties: our definitions are in terms of simple types such as 'integer' or 'character string', and a suite of operations whose behavior characterizes file properties.

The operations described in this section are not exactly those we will use in the service specifications of chapter 6; they are meant to be evocative of the kinds of operations a specific server will implement. We use a procedural notation that supports multiple return values; in some languages, such as C or **Pascal**, the actual implementation would be different.

4.6.1. Basic simple operations

The first two operations are implicit operations on property list groups, in that we presume that there is a one-to-one association between objects (for example, file histories or directory nodes) and property list groups. Thus, to create a property list group one must create an object; there is no identifier for a property list group except the object identifier.

CreateObject () ⇒ (Status, ObjectID)
Allocates a new object and creates an empty property list group for it; the group contains no property lists.

DeleteObject (ObjectID) ⇒ (Status)
Deallocates the object and deletes all of its properties and property lists.

The *ObjectID* returned by ***CreateObject*** is used to identify the object for all further operations. Non-property operations on objects do not concern us here, except to note that because they may implicitly affect property values (e.g., file length) they complicate formal descriptions of the semantics of property operations. The only way to do so is to precisely describe the side-effects of the non-property operations on property values, in terms of explicit property operations.

With an identifier for an object and its property list group, we can create property lists for the object:

CreateList (ObjectID, Property-List-Name) ⇒ (Status)
Creates a new property list with the given (string) name.

DeleteList (ObjectID, Property-List-Name) ⇒ (Status)
Deletes the named property list and all of the properties on it.

For multiple-version files, *Property-List-Names* are version identifiers. The file system implicitly creates a list with a distinguished name, to serve as the file history list, when the file is created, and the version lists might also be implicitly created, when new versions are created. Deleting a version list is synonymous with deleting the version itself; deleting the file history property list might be illegal unless no versions exist.

For directory nodes, creation and deletion of lists are the "enter" and "delete" operations for entries, respectively.

We can perform operations on specific properties of property lists:

PutProp (ObjectID, PropertyListName, PropName, PropType, PropValue, PropProtection) ⇒ (Status)
Inserts a new value binding for the property name; if a binding already exists, it is replaced.

GetProp (ObjectID, PropertyListName, PropName) ⇒ (Status, PropType, PropValue, PropProtection)
Returns the value currently bound to the property name, if one exists.

RemProp (ObjectID, PropertyListName, PropName) ⇒ (Status)
Removes the binding associated with the property name.

These three operations are necessary and sufficient to support arbitrarily large property lists.

4.6.2. Basic iteration operators

The operations described in the previous section are sufficient to manipulate properties and property lists, provided that one knows their names. However, because both name spaces are extensible, and because they are probably shared by a number of different applications and users, it is not likely that every property list name or property name will be known. We need a way to find out what names exist.

4.6.2.1. Iteration mechanism

There are several possible approaches to the retrieval of a list of names. One is to define an operation that simply returns all the extant names as one composite value, perhaps as a list or an array. Although this is simple to describe, it is not necessarily simple to implement; it requires that the communications mechanism be able to pass a large, and potentially unbounded, composite value.

Another approach is to provide a means of iterating over the set of names. Our model for this is an "iterator," based on the "generator" concept in Alphard [Shaw77] and CLU [Lisk77]. An iterator is an instance of an abstract data type that supports two operations: initialization, and "return next value." Thus, to obtain a list of N names, a client would have to perform $N + 1$ operations on the iterator.

Since remote operations are usually expensive, it might seem counter-productive to cast name-list retrievals as iterations, instead of as one-shot operations. However, while it is possible to construct a one-shot operation out of an iterated operation, the opposite is not possible. Iteration is more "primitive," and does not suffer from the "unbounded composite value" problem. In section 6.7.4, we show how to obtain good remote performance without sacrificing the ability to avoid unbounded transfers.

Many of the operations on property lists described in this thesis will be iterators. They all take the same form:

IteratorInit (Range-Specification) ⇒ (Status, IteratorKey)
Range-Specification typically identifies a name space over which to iterate, and possibly a selection predicate. *Iterator-Key* represents the state of the iteration.

IteratorNext (IteratorKey) ⇒ (Status, NewIteratorKey, Value)
Returns the next *Value* in the iteration, and a *NewIteratorKey* that is the successor to *IteratorKey*.

A client uses this pair of operations by obtaining an *IteratorKey* from the *Iteratorlnit* function, then repeatedly applying the *IteratorNext* operation, until it indicates that the last value has been returned. Because die state of the iteration is entirely encoded in the *IteratorKey,* there is no server state that must be preserved; hence, the server can crash and recover during an iteration without disrupting it.

We do not demand that the iteration functions impose any particular order on the values returned. Sorting a set of values can be done as efficiently by a client as by the server; requiring the server to return values in an externally meaningful order might make it expensive to record the entire state of the iteration in a compact *IteratorKey.*

4.622. Property list iteration

In order to list the entries in a directory, or to see what versions of a multiple-version file exist, one must be able to iterate over the set of property list names in a property list group. This is done using these two functions:

NameIterInit **(Object! D) =* (Status, IteratorKey)**
Initializes an iterator over the set of property list names for the object.
NameIterNext **(IteratorKey) => (Status, NewIteratorKey, ListName)**
Returns the next property list name in the iteration.

4.6.23. Property name iteration

In order to list the properties on a property list, for display to a user or as part of the process of copying an object, one must be able to iterate over the set of property names. Whereas the *NameIterNext* operation described above returns only the property list name, in this case it is better to return both the property name and the property value. Not only is this easy to do, since there is only one value associated with each property name, but it also optimizes the dominant case, since one will usually want to obtain the property value as well as the property name. Returning the value as part of the iteration means that the client need not use ***GetProp*** each time ***PropIterNext*** **is called.**

PropIterInit **(ObjectID, PropertyListName) => (IteratorKey, Status)**
Initializes an iterator over the properties on the specified property list.
PropIterNext **(IteratorKey) =>**
(NewIteratorKey, PropertyName, PropertyType, PropertyValue, PropertyProtection, Status)
Returns the next property in the iteration.

4.6.3. Composite operations

The basic operations described in the previous section can be combined to perform any desired operation on property values. However, logical sufficiency does not guarantee good performance. A practical system will include "composite," non-primitive operations where they reflect a more appropriate allocation of effort between server and client. Composite operations should be provided, in general, when it would otherwise be difficult or expensive to accomplish important functions. Performance is the justification for combining a number of simple operations into a complex one at the server, as opposed to in a library function at the client.

Performance in a distributed system suffers when communication is a bottleneck. Communicating a given set of data is almost always done more efficiently in one remote operation than in many. For example, if a client wants to retrieve all the properties of a file, it is more

efficient to do so in a one-shot operation, instead of using the explicit iterators described in section 4.6.2.1.

However, it is not always possible or practical to implement one-shot operations, especially if a simple RPC or message-passing system is used. Since these operations are purely performance enhancements, they can be implemented as library functions on client nodes if the server cannot provide them. In short, a composite operation should be provided at the server node if

- it provides a significant performance improvement over using the basic operations
- it will be used frequently enough to affect overall performance, or will be used where short response time is required
- it is not extremely difficult to implement.

Specific choices for composite operations depend on the details of an actual system design; in chapter 6 we will propose a set of composite operations for a real system.

4.6.4. Query operations

One class of composite operations that should almost certainly be provided at the server are *'query" operations: those which apply a selection predicate to a set of values and return the subset which matches. For example, one might formulate queries to return:

- The versions of a specific file that were never read after they were written.
- The names of Pascal source files in a given directory.
- All the executable program binaries in a file system that are linked with a given object library.

Each of these predicates can be evaluated by examining file properties. One could implement them solely in client-resident code by retrieving the relevant properties for all files in the domain of the query, then applying the predicate at the client. For small domains (the versions of a single file, for example) this is perhaps reasonable, but if the domain is "the entire file system," one can do much better by performing the selection at the server.

A large part of the performance improvement comes from reduced communication costs, since the size of the answer is probably insignificant compared to the size of the domain. However, one can also obtain dramatic gains by using more efficient search techniques. Performing the selection at the client involves examining every element in the domain. Performing a search at the server can often be done in time proportional to the size of the range, if the appropriate indices are maintained at the server.

There are three sizes of domains over which one might want to perform a query: property list groups (a single directory node or a multi-version file), entire server databases (a directory system or a file system), and multiple-server distributed databases. As the size of the domain increases, efficient implementation of queries becomes much more difficult. On the other hand, the potential benefit from implementing the query at the server also increases, provided that a favorable ratio of range size to domain size is maintained.

4.6.4.1. Query function abstraction

Queries are implemented as iterators, but unlike the simple "listing" iterators described earlier, they require specification of a property name and an explicit selection predicate. The encoding of the selection predicate is a detail which we will cover in chapter 6, but in general

terms, a query predicate specifies a property value type and subset of values. In chapter 6, we restrict these subsets to sub-ranges. If the type is "unordered," as described in section 4.3.1.1, then a "sub-range" must contain only one value; that is, the predicate must be an equality predicate. Otherwise, the sub-range is described by inclusive upper and lower bounds on the value (which may be equal). A predicate may include a negation operator: for unordered types, this provides a "not-equal-to" predicate, and for ordered types it allows selection of values outside of a range instead of within it.

We believe that subrange predicates are sufficient for most applications, although this restriction will make a few queries harder to perform. We do not make provision for conjunction, disjunction, and other complex queries (such as those involving aggregate values). Evaluating a disjunctive query should not take significantly longer at the client than at the server, since the ratio of the amount of information returned will be within a small constant factor. We suspect that most conjunctive queries can be optimized so as to eliminate all but a few candidates on the first selection; proof of this conjecture requires experience with actual systems. Intuitively, we expect that a conjunctive query with a small result will have at least one highly selective sub-query; the trick is to recognize such sub-queries and perform them first. (Section 11.3 discusses one mechanism for doing so; there are a variety of approaches to estimating query selectivity [Chri81].) Since simple queries will be dominant, we should not burden the design and implementation with handling complex queries.

4.6.4.2. Queries over property list groups

When the range of a query is a single directory node, or the set of versions of a single file, it can be implemented as a query over a property list group. A property list group is likely to be a relatively small database, so it is not necessary to establish auxiliary data structures to speed the processing of such queries; linear search is sufficient.

The query operations over a property list group are:

GroupQueryInit (ObjectID, PropertyName, QueryPredicate) =* (IteratorKey, Status)
Initializes a query applying *QueryPredicate* to the *PropertyName* properties on each property list in the group.

GroupQueryNext (IteratorKey) =» (NewIteratorKey, PropertyListName, PropType, PropValue, Status)
Returns the next value matching the predicate, and the name of the property list on which it appears.

4.6.4.3. Queries over entire systems

When the range of a query is an entire file system or directory system, performance requires that one avoid a linear search. Instead, the system must maintain suitable indices, so that one can quickly map a range of property values onto the set of objects (and property lists) which are bound to those values.

However, maintaining these indices is not without cost, in both storage space and processor time, for ***PutProp*** operations. (The existence of an index should not affect *GetProp* and similar "read" operations.) Because of this cost, it might be desirable to maintain indices only for certain properties. If so, the server interface must include a mechanism for establishing an index, and there must be a policy determining who may create an index.

The query operations over an entire system ("global queries") are:

GlobalQueryInit (PropertyName, QueryPredicate) => (IteratorKey, Status)
Initializes a query applying *QueryPredicate* to the *PropertyName* properties on each property list in the group.

GlobalQueryNext (IteratorKey) => (NewIteratorKey, ObjectID, PropertyListName, PropType, PropValue, Status)
Returns the next value matching the predicate, the identifier of the object, and the name of the property list on which it appears.

It might be necessary to translate the returned *ObjectID* into a directory path name for the object, to make the results of the query meaningful to a human. One way to do this uses "back-pointers" from objects to name contexts; once such a back-pointer is followed, a *GroupQuery* operation can be used to translate the *ObjectID* into a path name component.

4.6.4.4. Multi-server queries

In a distributed system, there may be several distinct file systems or directory systems[7]. A user might want to apply a query over the entire range of systems, not just a single system. Is is appropriate to provide special support for multi-system ("universal") queries?

The answer might seem to depend on the extent of the distributed system, but in fact it is "no" in any case. If there is a relatively small set of servers involved, then to perform a query over all of them one should performs individual global queries over each of them, and take the disjunction of the results. This can be easily done by a client-host library routine, which takes a list of the servers that should be examined, in addition to the property name and selection predicate.

If the set of servers is large, and distributed in such a way as to make communication expensive, then it might seem wise to try to avoid querying each one individually. However, any conceivable alternative, such as a centralized inverted index, involves significantly more communication during ***PutProp*** operations than would be required to query each host, unless universal queries are extremely frequent compared to ***PutProp***s. It does not appear useful to include specific support for universal queries.

[7]This means several logical file services; it does not mean one logical file service whose implementation is transparently distributed over several hosts.

Chapter 5
How Are Properties Used Now?

Before we proceed, in chapter 6, to describe our file system design, we briefly investigate how properties are used in existing systems. In section 5.1, we examine what facilities current systems usually provide to support file properties. In section 5.2, we look at how these facilities are used to solve various problems. In section 5.3, we summarize the property support facilities available in a variety of existing systems.

5.1. Traditional Facilities for Property Support

Almost every file system provides some support for file properties. Because these "traditional" facilities are so widely used, they are taken for granted as a feature of modern file systems. Such systems provide generally similar facilities, although not with uniform interfaces.

We look at four facets of traditional facilities: how properties are represented; how and when property names are bound; how extensibility is provided; and what operations on properties are supported.

We define a "traditional" file system as one that does not provide extensible property support, a uniform and abstract representation of properties, or an efficient property-based search mechanism. While there are systems that provide one or two of these features, most provide none, and none provide all.

5.1.1. Representation

Every traditional file system represents properties in its own idiosyncratic fashion. Worse, the representation is usually exposed as the only interface between user programs and file properties. This is a major obstacle to writing portable programs that use properties. Programs must manipulate properties by reading and writing a "file header block," a fixed-length record associated with each file. On some systems, such as the Xerox IFS [Schr85], the header block actually appears in the address space of the file, and is accessed with the normal read and write file system calls. Other systems, such as TOPS-20, provide special system calls that read and write the header, or, as in Unix, that convert an internal representation of the file header to an external one.

To protect certain fields from indiscriminate modification, a system may intercept writes to the file header and update only specific fields. Other systems do not provide a way to write the entire header, but instead present several system calls, each used for updating a specific field. For example, in Unix there are separate calls to change the owner, the protection group, the last-accessed and last-modified times, and the "mode" of the file (which includes bit fields both for access control and other properties).

Because these file system calls are conceived of as providing access to a file header block, instead of to an abstract data type, the data formats used are specific to the particular implementations. Disk space is assumed to be scarce, so fields in file headers are compressed as much as possible; this means that often they are neither full-size integers, nor aligned on natural boundaries. Boolean properties are coded as single bit fields at arbitrary offsets.

These "naked" representations have unfortunate consequences. They complicate, and make unportable, programs that access properties, because there is no uniformity in the representation. More important, they discourage any reimplementation that would be incompatible with previously published formats.

One system that does present an internally uniform interface to file properties is VAX/VMS [Digi83]. In VMS, although some "property values" are in turn arrays of bits encoding several booleans, each property may be accessed individually. Moreover, all properties are accessed by a uniform mechanism: instead of being a field in a data structure containing many properties, each property is identified by an arbitrary integer. The additional level of indirection insulates user programs from details of the implementation. Unfortunately, the VMS mechanism is still too idiosyncratic to permit portable programs.

5.1.2. Property Name Binding Time

File systems vary in how they map property names onto values. We can distinguish a spectrum of possible binding times, and observe how delayed binding trades efficiency for flexibility.

In the simplest systems, those that present file headers (or facsimiles) at the system interface, binding is at compile-time, because each property value is at a fixed location in a data structure. There is no run-time cost for binding, but there is also no flexibility; the file system implementation cannot be changed without recompiling all programs.

In a system such as VAX/VMS, the binding between a property name and its integer identifier is still normally made at compile time, but the binding from the integer identifier to the value is done at run time, perhaps by a table-lookup. While the run-time cost is slightly higher, the file system implementor is free to rearrange the data structures. Still, client programs must be recompiled to take advantage of a newly-defined property.

The latest possible binding time is when a string representing the property name name is passed to the file system at run time. This is relatively expensive, since mapping an arbitrary string to an address is not a trivial operation. It costs both processor time, to perform the mapping operation, and storage space, since a database is required to maintain the mappings. Nevertheless, late binding is the only way to support the addition of arbitrary new properties without recompiling user programs, and is used in some existing systems, such as the Symbolics LISP Machine [Symb85].

5.1.3. Extensibility

One hallmark of traditional file systems is their support only for intrinsic properties, or at best a small, unextensible set of additional properties. This is probably because implementation of extensible property support requires both late binding and dynamic storage allocation, and is therefore avoided as too costly or difficult. The property support in traditional systems is not useless, since those properties that are supported are in all probability the most important ones. Often, however, users of an unextensible file system need to represent unanticipated properties.

How Are Properties Used Now?

Not surprisingly, a number of *ad hoc*, often inelegant, solutions are used. They all involve finding additional places to stash property values, and fall into four categories:

File name conventions

A file name can be used to encode property values. Typically, the name is broken into two parts, and the "extension" part is used to encode the file type. For example, a language processor on TOPS-20 would assume that `BLAISE.PAS` is a Pascal source file. There are good psychological reasons for using such encodings as "hints," but this is not an adequate replacement for extensible property support.

Embedded properties

Properties can be stored in the contents of a file. For example, the first few bytes of a file might specify its type. When this can be done, it requires no support from the file system, and can store arbitrary amounts of information, but often it cannot be done: when some users of the file are not aware of the convention distinguishing "properties" from "contents"; when it is desirable to separate access to the file from access to its properties; when the size of both the file and its properties varies frequently, etc. Even when it is possible, the convention inevitably varies from file to file, so the mechanism for accessing the properties might be reimplemented many times[1].

Recycled properties

Intrinsic properties that can be modified by the user, and whose "official" values are not relevant to the application, can be used to encode new properties. On Unix, in lieu of a property such as "NoMessagesToTerminal", the "executable by owner" protection bit for a terminal is used as a flag to suppress asynchronous messages from other users. Since terminal "files" are never executed as programs, this bit is not otherwise significant.

Recycling is inelegant, because it ignores the normal semantics of the property, and it is only rarely possible to find a property to recycle.

Auxiliary files

When the "property" information cannot be represented otherwise, it can be stored in an "auxiliary" file. For example, a mail-queuing system may store a message in one file, and the destination of the message in a small additional file.

Auxiliary files are clumsy because they are accessed via a different mechanism from that used to access intrinsic properties. Also, they are not firmly attached to the files they describe; utilities that move, copy, rename, or delete files must be taught to look for auxiliary files, or the connection between the file-pairs will be lost.

There are, then, a number of unsatisfactory, but sometimes serviceable, ways to store "user-defined" properties even in traditional systems. Since all are based on features of a particular file system or data file structure, they are inherently unportable and irregular. File name conventions and recycled properties are quite limited in expressive power, and do not store the names of extended properties. Embedded properties are unsuited to many file formats, and auxiliary files do not reliably attach a file to its properties. Both embedded properties and auxiliary files require some parsing to manipulate properties, which decreases performance.

---

[1]The Symbolics LISP Machine [Symb85] has a system-wide convention for embedding properties in text files.

5.1.4. Traditional Operations

Traditional file systems usually support just a few operations on properties. Typically, one can modify the value of certain specific properties. Some systems allow clients to read individual properties; others provide an operation that reads all the properties at once. In addition to these explicit operations, a file system performs implicit operations on a file's properties when a client performs explicit operations on the file. For example, it might update the "Length" and "Modification-Time" properties whenever a client writes to the file, and might implicitly read protection information for a file when a client attempts to open it.

Since the set of possible properties is fixed, there is no need to provide an operation to list the properties of a file; one can always find the list in the programmer's reference manual. There are, of course, no operations to add new properties or remove existing ones.

5.2. Traditional Uses of Properties

In spite of the limited property support in traditional systems, we can learn a lot by surveying how properties are used in these systems. The points of strain, where traditional systems are not up to the job, show why better support is needed.

We find several classes of uses for traditional properties. Because properties in traditional systems are intrinsic, the dominant use is for communication between client and file system. Three other uses support communication between clients: properties are used to record a relationship between two files, or by programs to determine the right approach for processing a file, or to select a group of files based on some query.

5.2.1. Communication between user and file system

The most basic use of file properties is to record information of interest to both the client and the file system. The file system, through implicit operations, maintains property values to reflect the state of a file. Its length, or the time it was last modified, may be important to a client, and these properties serve as a way for the file system to communicate information to the client. This information is frequently presented by a directory-listing program (although that can be a misnomer, since property information is often not stored in the directory).

Directory listing and program-initiated reading of properties is one-way communication. Users communicate information in the opposite direction through explicit modification of intrinsic properties. For example, if access-list protection is used, the owner of a file can set the protection property. If a file system supports the notion of a temporary file, a client can mark a file as temporary.

Many file systems support a rudimentary concept of file type, to communicate information about the structure or intended use of a file. A file system that includes an integrated directory system, such as Unix, needs to distinguish between files that are directories and those that are not. Since a content-based check may be unreliable or expensive, Unix has a special property to denote a directory. Other systems, such as IFS [Schr85], distinguish between binary and text files. Although such distinctions may not be intrinsic to the operation of the file system, they can be important to file system utilities; for example, LOCUS [Pope81] distinguishes mailboxes from other files, and uses this distinction when recovering from network partitions.

5.2.2. Relationships between files

One common use of file properties, which although used in almost all systems paradoxically is supported in almost none, is to record relationships between files. Usually, these relationships are denoted by use of similar file names, according to well-known conventions. So firmly established is this convention that most TOPS-20 users would be surprised if a file named `PROGRAM.PAS` were not the source for an executable file named `PROGRAM.EXE`. We assume that files with names differing only in their suffixes are related, if they appear in the same directory.

The number of properties that can be represented this way is limited: it is not clear if a file named `PROGRAM.OLD` is an old version of the source file `PROGRAM.PAS`, or the executable file `PROGRAM.EXE`. While systems such as Unix allow names such as `program.c.old`, ameliorating this particular ambiguity, it is still cumbersome to encode multiple types this way.

Moreover, because the relationship denoted this way is conventional, not formal, it can only be used in certain circumstances. For example, many-to-one relationships (e.g., many source modules for one program) are hard to represent, many-to-many (e.g., many sources modules shared by many programs) harder still, and relationships between files in separate directories are tenuous. It is better to use name-borne information as a "hint," for the benefit of rapid searching, and use formal properties to represent formal relationships. We discuss this in more detail in section 11.5.

5.2.3. Programmed manipulation of files

Many applications programs that operate on files use properties of these files to determine how they should be treated. This is why file systems must provide property access functions, rather than relying on "commands" or "system programs" as the only way to access properties.

A common application that makes good use of file properties is the automation of the compilation process; for example, the *COMPILE* command on TOPS-20, or the Unix *make* program [Feld79b]. *Make* is given a description of the modules that form a program, including a list of dependencies between source and object files, and instructions for each step of the compilation process. Using this description, *make* then produces those files that do not exist. In the absence of additional information, *make* would have to recompile everything to ensure that the final program was up to date. However, using the "last modified time" property of the files, it can avoid redundant work, by combining this information with the dependency graph to know which source files must be recompiled.

Sometimes a program will accept several different types of input file, taking slightly different action for each type. Although the the user could specify the file type as part of the command, this can complicate the user interface. More important, the user may be mistaken. A rudimentary type check will prevent many disasters, and one might as well use the type check to automate the procedure.

For example, a program used for sending documents to a print server might accept both text files and intermediate-format files. It could try to distinguish the two file types based on the form of the input file name, but this is easily frustrated by a unconventionally named file; the result might be that a pre-formatted file is treated as a text file, and a great deal of paper wasted. One could embed a special code in all pre-formatted files, one that is unlikely to appear in any other kind of file, but there is no guarantee that this code will never appear. The best solution is an explicit "type" property, set by the formatting program.

5.2.4. Searching for files

Users often need to search a large file system for a file or set of files. It may well be the case that the files must be found even though their names are not known; the search must be based on some other attribute.

A variety of predicates may be used to select from a set of files. For example, the contents of the files might be checked against a pattern, or a specific portion of the contents might be tested. Frequently, files are sought using a test on property values. Other searches match the names of the files against a pattern; since properties are often encoded in file names, checking names against a pattern may actually be a property-based search.

To search a small context in a traditional systems one usually obtains a "directory listing" (which may list file properties as well as file names), and then scans it visually to find the desired files. Often, a simple filtering program, such as a string pattern matcher, is used on the directory listing to reduce the visual search space. Searches carried out entirely under program control may be more direct, but the mechanism is often re-implemented in each program.

Because it is such a common operation, many systems provide built-in pattern-directed searching for file names. For example, on Unix one could search for all the C source files in the current working directory using a filtering program[2]:

```
ls | grep '\.c$'
```

but it is much simpler and faster to use file-name pattern-matching:

```
ls *.c
```

Unfortunately, this facility is not efficient for global searches.

Other local searches involve intrinsic properties; for example, one might look for:

- All the Pascal source files
- The oldest (or youngest) files
- The largest files
- The executable files
- The subdirectory files

within a directory.

Global searches, although costly, are still occasionally necessary. It is impractical for a user to "visually filter" a listing of all the files in a large file system, so some automation is required. File-name pattern-matching can be done over an entire system using a directory-listing program that recursively descends the directory hierarchy, then filtering the resulting list through a string pattern-matcher. Sometimes, the list is "pre-computed," say once each day, which makes such searches more efficient but less current.

When a global search is based on intrinsic properties, the only method normally available is to use a program that examines the properties of each file in the system. On Unix systems, a program called *find* exists for this purpose. *Find* searches a sub-tree of the directory hierarchy, or the entire hierarchy, and selects files based on queries constructed from tests on file name patterns or intrinsic properties. A common use is to remove all the editor checkpoint files (identified by file names ending in `.CKP`) more than three days old; this is done once a day to reclaim storage space.

Ultimately, the recursive-descent approach to global searching founders, because it in-

---

[2]Note that under Unix, file-name pattern-matching is done by the command interpreter, not by the directory system. This makes most programs unable to use file-name patterns unless they are presented on the command line, and leads to an inconsistent user interface.

herently takes more time as the file system grows larger. For example, on a typical Unix system with about 70000 files, it takes 20 minutes (on a VAX-11/780) just to list all the file names. Global searching can only be a convenient tool if the search time depends on the number of matching answers, not the number of files to be tested.

5.3. Property support in existing systems: A survey

We conclude this chapter by summarizing the file-property support in a variety of existing file systems. Table 5-1 characterizes most of the file systems mentioned in this chapter and in chapter 3. For each file system, we indicate whether or not it has extensible property support, whether every property is accessed the same way, the method by which properties are accessed, and for non-extensible systems, the number of non-intrinsic properties. There is no column labeled "supports efficient property-based search" because no extant system does so[3].

The table also lists a few file systems not described elsewhere in this thesis. The Carnegie-Mellon Central File System (CFS, not to be confused with the Cambridge File Server) and its successor Sesame, are interesting because they support several non-intrinsic properties. One is an "Advisory File Type," that is stored but not interpreted by the file system. CFS also supports an "Advisory Semaphore"; this is not entirely non-intrinsic, since the server will clear the semaphore when a specified timeout expires, but does not otherwise manage it. Both of these advisory properties are meant for use by higher-level applications.

Several systems have unusual file property facilities. The Univac-1100 [Univ85] directory is actually two address spaces. One maps (*file-name*, *type*, *subtype*) triples to file identifiers; *type* and *subtype* are integers. *Types* are taken from a small set of system-defined codes; some *subtypes* are user-defined. The other address space maps arbitrary symbols to arbitrary strings.

The Apple Macintosh [Appl85] uses properties in an unusual way. In some systems, a file may have as a property a back-pointer to a directory that refers to the file. Macintosh files have these properties, but the forward pointers are missing; the Macintosh directories ("folders") do not mention the files they contain. This is feasible because the Macintosh file system holds few files.

[3]The Xerox Filing Protocol supports property-based search, but only within a directory and its descendants.

System Name	Extensible	Uniform	Access method	Number of Non-Intrinsic Properties	References
Alto	yes	no	leader page (1)	extensible	[Thac82]
CDFS	yes	no	(2)	extensible	[Garf85]
CMU CFS	no	no	(3)	5	[Acce80]
CMU-ITC Vice-I	yes	yes?	(4)	extensible	[Saty85]
CMU-ITC Vice-II	no	yes?	(4)	none?	[Saty85]
INTERLISP/ File package	yes	yes	GETPROP/PUTPROP	extensible	[Xero83]
INTERLISP/ File system	no	yes	(5)	0	[Xero83]
Leaf	no	no	leader page (1)	0	[Mogu81]
NSW file package	no	not really	catalog name (6)	0	[Cash76]
Sesame	no	no	(7)	2	[Thom85]
Sun NFS	no	no	stat-like (8)	0	[Sand85]
Symbolics	yes	yes	(9)	extensible	[Symb85]
TOPS-20	no	no	GETFDB/CHFDB (10)	0	[Digi80]
Univac 1100	yes	no	directory (11)	extensible	[Univ85]
Unix	no	no	stat (12)	0	[Ritc78]
VAX/VMS	no	yes	$QIO option (12)	0	[Digi83]
Xerox Filing Protocol	yes	yes	(14)	extensible	[Xero85b]

Notes on "Access Method":

(1) Leader page appears in file address space; no specific property operations.
(2) Property and file header operation definitions missing from [Garf85].
(3) Various operations are used to set specific properties. *GetHeader* used to retrieve entire file header.
(4) File system operation, not described in detail in [Saty85].
(5) GETFILEINFO/SETFILEINFO functions, similar to GETPROP/PUTPROP.
(6) "System attributes" are stored as part of a file's name in the NSW file catalog; the encoding would support extensible properties.
(7) Properties are set using several flavors of the operation that writes a file. *GetFileHeader* used to retrieve entire file header.
(8) Entire file-status structure read by *getattr* operation; *setattr* operation writes all modifiable fields.
(9) **fs:file-properties** function returns all properties of a file; **fs:change-file-properties** changes or creates a set of properties.
(10) GETFDB returns the entire file descriptor block (FDB); CHFDB modifies a set of FDB fields.
(11) The directory is used to map from symbol names to strings.
(12) Entire file-status structure read by *stat* operation; modifiable fields written by *chmod*, *chown*, and *utimes* operations.
(13) A list of attributes to be read or changed can be specified for several functions of the $QIO system call. (Only partially uniform; some attributes are complex structures.)
(14) **ChangeAttributes** used to create or modify attributes. **GetAttributes** used to read attributes; other functions use attributes to specify particular files.

Table 5-1: File-property support in existing file systems

Chapter 6
A File System Design

In the preceding chapters we presented a model of file properties, and described how file properties are used in traditional systems. In this chapter, we propose a design for a real system, including both directory and file service, that takes the general design implied by the model in chapter 4 and fills in the details necessary for an implementation.

This design is a "paper exercise," since the implementation described in chapter 7 covers only a small part of it. This does not mean that the design is an automatic translation of the model into a set of functions. By specifying the actual behavior of a complete set of functions, we can see if the model has holes, or is internally contradictory. We can also see if the resultant system seems to make sense, or if it is cumbersome, complex, and inelegant.

Since this is a design for a system, we must necessarily make concrete decisions on choices that are left open by our model. This does not mean that these decisions, such as the names of intrinsic properties, are the only ones possible; this is simply one specific point in a space of designs.

6.1. Goals and limits

The design presented in this chapter is not meant to be used under a particular operating system. Instead, it is meant to be useful for a wide class of moderately sophisticated systems, incorporating at least:

- Message-based or RPC inter-process communication.
- Support for multiple processes per program.
- Sufficient address space and memory to make "squeezing" the implementation unnecessary.

We are not trying to design a system that will require minimal disk storage space and processor speed. An implementation should minimize these requirements for a given design, and the design should not preclude efficient implementation. On the other hand, we want to provide more functionality than existing systems, and there is an inevitable cost to this. Thus, our design is aimed at systems composed of modern, low-cost hardware, so that one can afford to spend a few per cent of the system resources on increased overhead.

Since we are interested in distributed systems, we will also assume that the target system provides a high-speed Local Area Network (LAN) connection between hosts. Systems connected only over low-speed networks may benefit from this design, but they are not the primary target

A number of issues arise in the construction of a distributed system that, while central to the success of the system, need not be solved in this thesis. These include reliable, secure com-

munication, crash recovery mechanisms, and user authentication. These will be discussed briefly in the following sections, but we will take their solutions for granted.

6.2. Communication mechanism

The file and directory services described here are meant to be used in a distributed system, but they are also meant to be useful in an isolated computer system. We therefore describe the interfaces in terms of procedure calls.

We expect that an efficient Remote Procedure Call (RPC) mechanism, such as described in [Birr84], is available for our distributed system. Our requirements for an RPC include:

- **Multiple "connections" or connectionless communication.** An RPC system limited to one connection at a time will not be useful with a collection of directory and file services.
- **Dynamic binding.** It must be possible to bind an interface to a server exporting it at runtime, since the collection of servers might not be known at compile-time.
- **String data types.** The RPC system must be capable of communicating character strings of moderate length.
- **Return of large data types.** Character strings and large arrays of bytes must be allowed as returned values. Large arrays are necessary to implement *write* and *read* file system calls efficiently.
- **Multiple return** values. Many of the procedures defined here will return multiple values of various data types. The RPC can either support multiple return values, or return of heterogeneous structures that include string components.

Other desirable features are exception reporting, unbounded data transfer, and dynamic array types. These all contribute to efficiency and ease of programming by obviating the need for continual status checking, complicated buffer schemes, and maximal-size buffer allocation.

Since one client process can be bound to several instances of an interface it has imported, the call operation must allow specification of a server instance. This can be done either with an additional parameter, or in some languages by using a syntactic mechanism to specify the interface instance.

6.3. Protection

Because files and directories are the primary means of sharing in a distributed system, they are also the focus of protection issues. This thesis is not about protection mechanisms, but they are so intimately connected with the design of a file system that we must address them.

We take a simple approach to protection, both to avoid irrelevant detail in this chapter and because a secure protection system for heterogeneous systems is beyond the scope of this thesis. The protection system should prevent a malicious user from destroying the data of another user, modifying or reading the data of another user without authorization, masquerading as another user, or tying up shared resources. At the same time, the protection system should not interfere with controlled sharing.

6.3.1. Authentication and encryption

We require a reliable way to verify a user's identity. We assume that an "authentication server" exists, with which users are registered and that is willing to certify that a (*user name*, *password*) pair is valid. The simplistic scheme we use in this chapter, in which a client presents a plaintext password to the file server, which then asks the authentication server to validate the password, is far from secure; there are a number of simple improvements to this scheme, involving encryption and handshaking, to make it much safer [Denn82, Sing85].

Not only must the server be able to establish the client's identity (and vice versa), but communications between them must be secured against eavesdropping and spoofing. One approach is to encrypt the data stream. This can be done at several levels; it should be done in the RPC layer, to preserve the integrity of the connection (since we do not wish to reauthenticate the client for each command, it is necessary to protect temporary handles from eavesdroppers). It may also be necessary to encrypt the contents of files if clients do not trust the file server. One must balance the expected costs of penetration with the obvious cost of encrypted communication; in a university environment one might prefer to trust the community.

6.3.2. Protection model

Access control is one of the areas where existing systems differ widely. Our design is based on a simple protection model which supports most circumstances well, but does not attempt to solve the problem completely.

Capability-based systems have achieved some vogue [Wilk79], but they do not appear to be applicable to distributed systems with loose administrative coupling, since it is hard to prevent unauthorized copying of capabilities. We instead base our system on an access matrix, with columns denoting access rights and rows denoting actors.

6.3.2.1. Actors

We define an actor as either a *user* or a member of a wider superset. A user can be a member of one or more *groups*. One can also identify the set of users within a single administrative domain, and the universal set of all users. Traditional, single-computer systems do not always distinguish "administrative domain" from the universal set of all users, but in a distributed system where user authorization is carried out by several administrations, it can be necessary to distinguish between trusted and untrusted authorization domains.

A file or other protected resource, in our scheme, is owned by exactly one user and exactly one group, and exists within exactly one authorization domain. To represent the situation where a file should be owned by the union of several groups, we simply define a new group which is that union. It is harder to represent the situation where a file should be owned simultaneously by several users, if this is in fact a meaningful notion. An approximate solution is to create a new group comprised of these users, and designate one of them as the primary owner.

6.3.2.2. Access rights

Having defined the labels for the rows of the access matrix, we must also define the column labels: what rights can be applied to the various objects and properties in the system. Unfortunately, the sets of meaningful operations on files, file histories, directories, property lists, and properties all differ slightly, and so the sets of rights also differ.

Table 6-1 gives interpretations for the access rights "read," "write," "delete," and "append" for these five types of protected item. In some cases, the meaning of the access right name is stretched a little, but overall the assignments make sense.

A few explanatory notes for table 6-1:

- There is an asymmetry in some of the table entries. For example, to create a new property one needs access to the property list in question, but to remove an existing property one needs access to the property itself. This asymmetry is inevitable; a little thought to the alternatives should convince the reader. Also, notice that the right to apply ***putpropQ*** to an existing property is separate from the right to create a new property using ***putpropQ.***

- Property protection is protected separately from property value. The WRITE access right controls access to the property value; the APPEND access right (a slight misnomer) controls access to the property protection. Note that while APPEND access is a proper subset of WRITE access to file contents, APPEND and WRITE access to properties do not overlap.

- Contrary to common practice, we do not use protection to control the ability to "execute" a file. Executability is a *type* of a file, and should be represented as such. Treating executability as an access right separate from "read" access (to support proprietary programs) makes little sense in an distributed system, with multiple administrative domains. The appropriate solution is a "proprietary program server" that can be used to execute a program owned by another user on behalf of a client without read-access to the file[1].

- We want to be able to allow a client to change the value of a property but not its type. Therefore, we prohibit ***PutPropQ*** from changing the type of a property. To change the type of a property, a client must first delete it with *RemPropQ*, then recreate it. This procedure would get around any other kind of protection on property type, so the right to remove and recreate a property implies the right to change its type.

Protected Item	Protection Access Right READ	WRITE	DELETE	APPEND
File Contents	read()	arbitrary write()	.....	append-write()
File History	list versions	.....	delete file	create version
Property List	list properties	.....	delete entry or version	new property putprop()
Property	getprop()	modifying putpropQ	remprop()	change protection
Directory	list entries	.....	delete node	create entry

Table 6-1: Meaning of protection access rights for various items

If an actor is not explicitly allowed access, but the same access right is allowed to a larger

[1] **A obvious extension to this server provides the functionality of the Unix "setuid* • mechanism [Ritc78].**

protection domain of which the actor is a member, then the actor implicitly has the specified access.

6.3.3. Representing access rights

The internal representation of a protection attribute depends on the implementation, but we must have a canonical representation to use for communicating protection between client and server. There are two possible approaches: to encode protection as a string of characters, or a structure containing (effectively) a two-dimensional array of boolean values.

A string representation has the advantage that it is directly understandable by humans. We can use 'R', 'W', 'D', and 'A' to stand for the access rights "read," "write," "delete," and "append," and 'U', 'G', 'O', and 'E' to stand for the actors "user," "group," "organization," and "everyone." For example, the string "U:RWD, G:RW, O:R, E:" specifies: read, write, and delete access for the user who owns the item, read and write access for the owning group, read access for other members of the owning organization, and no access for everyone else.

For programs, representing access rights as an array is much more natural, because parsing is avoided and communication costs are lowered. Parsing and formatting functions can be provided at the human user-interface, so there is no need to maintain a human-readable representation at lower levels. For communicating protection attributes between clients and servers we will use the array representation described in section 6.5.2.

6.4. Identifiers

Operations on files and directories must, by their nature, constantly refer to servers, objects, and pieces of objects. In the design of a distributed system, the way things are identified is significant.

6.4.1. Objects

In a distributed system with multiple servers, one must be able to identify potentially migratory objects, such as files. If the object identifier includes its location then finding the object is easy, but migrating it is hard. If the object's location cannot be directly determined from its identifier, then migration is easy, but locating the object (the more common task) is hard.

One way to resolve the location/migration tradeoff to assign each object an invariant, globally-unique identifier, but associate with it a "hint" that is likely to give the current location. A hint provides an easy way to find an object that hasn't moved, and does not prevent use of more sophisticated methods to locate an object that has moved. This sort of approach is discussed in [Dala81].

In our system, each object is assigned a globally-unique identifier (GUID) that never changes. Whenever a GUID is stored as a reference to an object, we also record the server that is believed to currently manage it. If this server identifier turns out to be wrong, then an "object locator service" could be invoked to track down the object, and the server hint is corrected. A similar approach is used for process migration in DEMOS/MP [Powe83].

We follow the approach described by Birrell and Nelson [Birr84] for naming servers. A

name server such as Grapevine [Birr82] is used to store bindings between a server instance name and an address suitable for use by the communications mechanism. We always store server identifiers as strings.

6.4.2. File Versions

Since our file system supports multiple versions of files, we need a way to identify a specific version of a file. Most multi-version file systems use a positive integer as a version number; distinguished values such as 0 or −1 are sometimes used to refer to the current and oldest versions.

Using integers, while simple, confines the idea of "version" to a simple linear sequence. File versions often form tree structures rather than such simple sequences. For example, a software product might have major and minor version numbers, with work proceeding simultaneously on versions 5.0 and 4.3. The SCCS [Roch75] and RCS [Tich82] systems, used with Unix to manage source files, support this non-linearity, and allow users to assign version identifiers composed of arbitrary tuples of integers.

Other systems impose even less restriction on version identifiers, allowing an arbitrary character string. An inelegant example of this is the MIT ITS file system, in which file names have two relatively short components. Normally, the second component is used to denote file type, as is traditional with "extension" fields, but it can also be used to denote a version. The system provides little support for this, except that the distinguished strings ">" and "<" respectively denote the "highest" and "lowest" versions of a file, according to a complicated scheme.

In our file system, there is no server-enforced mechanism for determining which version is "lowest," "highest," or "next" (i.e., next to be assigned). In particular, there is no syntax for such notions. However, we do include a server function to find the highest existing version in a sequence, to encourage the use of a common convention on version identifiers.

6.4.3. Temporary handles

Distributed systems designs should minimize the amount of state shared between client and server. Maintenance of shared state requires robust communication protocols, and careful attention to communication failures or host crashes. An ideal design would require no state maintenance at the server between transactions; the server could crash and restart without affecting the operation of the client. For example, the Sun NFS file system [Sand85] includes no ***open***() or ***close***() operations.

If the file server is stateless, every ***read***() call must pass the GUID of the file involved, and identification and authentication information for the client. This creates wasteful overhead by repeating operations that need only be carried out once: translation of the GUID to an internal address for a file descriptor; verification of the user's identity and access rights to the file; and allocation of temporary resources, such as data buffers, to the transaction.

By maintaining some shared state as a "hint," we can dramatically improve efficiency (and reduce clutter in the description of file system operations). Use of an ***open***() call allows the file server to do these expensive operations once, then denote the results with a shorthand identifier:

an open file descriptor[2]. Subsequent ***read***() calls pass this open file descriptor, reducing server overhead and avoiding the cost of communicating redundant information.

If the server crashes between transactions, the client can repeat the ***open***() call and then continue with its operations, since no unreconstructible state is lost. (Recovery from a crash during a transaction is in any case complex; one approach is described by Lampson [Lamp81].) If the client crashes, the server can discard the allocated resources after a reasonable period; if the client then recovers, it can reissue the ***open***() call and continue. The ***close***() call is not strictly necessary, but is an important courtesy on the part of the client, since it releases the allocated resources as soon as they are no longer needed.

In our system, temporary handles (such as open file identifiers) are represented as 32-bit integers, which are meaningless to the client and to other servers. A temporary handle can be used only on the server that issued it.

6.4.3.1. Locks on handles

We provide a simple locking mechanism, with the expectation that more complex schemes will be implemented using a separate lock-server process. For example, a lock-server could acquire a lock on a file and then mediate operations for all other clients, providing whatever level of locking is necessary.

We use the temporary handle to denote the holder of a lock on a file, directory, or part thereof. A lock is acquired on an open handle using a ***Lock***() operation, and is released using ***UnLock***(). While a lock is held, only the handle on which it is held can be used for operations on the locked item. Other handles for the same entity cannot be used without error, although they may be opened or closed.

There are three kinds of locks in our system. In order of increasing "granule size" or *lock level*, these are:

1. **File content locks**: used to lock the contents of a file version (these do not exist in the directory system).
2. **Property list locks**: used to lock a single property list: a version's property list in the file system, or an entry's property list in the directory system.
3. **Property list group locks**: used to lock an entire object: a file history or directory node.

Property list locks and property list group locks are incompatible if the objects that they lock overlap. For example, although locks may be simultaneously held on two different directory entries, there cannot be simultaneous locks on a directory entry and the entire containing directory node.

A file content lock is compatible with a lock on the associated version property list, but not with a lock on the entire file history. At most one type of lock may be held on a single file handle, so to lock both the contents of a file and its version property list, one needs two file handles.

In a distributed system, a client holding a lock can crash, and thereby fail to release that lock. Servers must guard against dangling locks by requiring periodic activity on the locked item. If the holder of a lock fails to use the locked temporary handle within some period, the lock is said to be "timed out."

---

[2]Authentication information can be safely denoted in this manner if the communication path is secured, perhaps by means of encryption.

A lock is not automatically broken when it times out. However, any attempt to acquire a lock conflicting with a timed-out lock will succeed; at that point, the original lock is broken. Further use of the timed-out temporary handle will result in errors, unless the client acknowledges that timeout by explicitly releasing the broken lock.

When a lock attempt succeeds by breaking a timed-out lock, the successful client is nonetheless notified that it is breaking a lock; it may be that the locked item is not in a consistent state. Repairing this state is not the server's responsibility. Further, if the server crashes, then all locks are broken. Thus, the servers described in this chapter do not directly support failure-atomic transactions (as defined by Spector and Schwarz [Spec83]).

Since all locks eventually time out, the servers do not attempt to detect or prevent deadlock.

6.5. Data types

One of the goals of the property abstraction is to minimize the problems associated with implementation-specific encoding of data types. To meet this goal we keep the set of data types small and straightforward.

Although abstractly these types are strings of bits, for practical purposes we must choose a canonical byte size for the distributed system as a whole; processors that do not use this byte size will have to perform a suitable conversion. Eight-bit bytes (''octets'') are standard for most network protocols[3] and most processor architectures, so all data will be communicated as streams or arrays of octets.

Quantities, such as integers, that are represented as more than one octet are to be transmitted with the most significant octet first. This conforms to the usage of the Internet Protocol family [Post81a] and others. Hosts that use a different byte-order internally must byte-swap multi-octet values [Cohe80].

6.5.1. Simple types

''Simple'' data types are scalar values without any structure. Except for real (floating-point) numbers, they can be treated as integers of various sizes, for purposes of comparison and assignment. We will use 32-bit integers, since this is a natural integer size for modern processors. 16-bit integers are too small for such quantities as the number of bytes in a file[4].

---

[3]For example, the DARPA Internet Protocol (IP) family [Post81a], IBM's SNA [Guru84], and the ISO standard protocols [Inte84].

[4]Larger integers will probably be necessary in the not-too-distant future, as single files reach multi-gigabyte size. Many systems will collapse at that point.

The simple types used in our system are:

Integer	A 32-bit two's complement integer.
Unsigned integer	A 32-bit unsigned integer.
Real	A 32-bit floating-point value in IEEE-754 [IEEE85] format. Although a larger format would provide more resolution, it is unlikely to be worth the extra storage costs.
Boolean	An integer which, if zero, denotes false, and otherwise denotes true.
Time	An unsigned integer representing the number of seconds since 00:00:00 UTC on 1 January 1970. Conversion to and from local time, for user-friendly interfaces, must be done by application software, at as high a level as possible.

With the exception of ***Boolean***, these are all ordered types; that is, it is possible to determine of any two values which one is larger.

6.5.2. Protection

Protection "values," as described in section 6.3.3, are two-dimensional arrays of booleans. One can consider the array indices to be values of the enumeration types {"read", "write", "delete", "append"} and {"user", "group", "organization", "everyone"}. The C structure declaration in figure 6-1 serves to define the order of bits within a Protection value; a true (non-zero) bit means that the specified access is allowed.

```
struct ProtectValue {
        unsigned short          /* the whole thing fits in 16 bits */

        /* Owning User rights */
            UserRead:1,
            UserWrite:1,
            UserDelete:1,
            UserAppend:1,

        /* Owning Group rights */
            GroupRead:1,
            GroupWrite:1,
            GroupDelete:1,
            GroupAppend:1,

        /* Owning Organization rights */
            OrganizationRead:1,
            OrganizationWrite:1,
            OrganizationDelete:1,
            OrganizationAppend:1,

        /* Everyone rights */
            EveryoneRead:1,
            EveryoneWrite:1,
            EveryoneDelete:1,
            EveryoneAppend:1;
};
```

Figure 6-1: Protection Value Representation

6.5.3. Character string and atom types

As discussed in section 4.3.1, we have four different ways of viewing character strings, depending on whether or not they are case-sensitive and on whether or not they are ordered. All four types are represented identically, as a sequence of ASCII characters with the left-most character first. Strings should include only include printing characters and white space, so as to be easily translated to other character sets.

6.5.4. Property values

A property value, since it can be one of a several possible data types, is represented as a discriminated union, or variant record. The record has three fields; the first is the discriminant field, the second a length field, and the third field is the value.

The discriminant field is a 16-bit integer whose value comes from the following mapping between data types and integers:

1. Integer
2. Real
3. Boolean
4. Time
5. Protection
6. Atom
7. Case-sensitive Atom
8. String
9. Case-sensitive String

The zero value denotes "No type." Values with this type are normally illegal and cannot be bound to a property name; however, they are used in certain query operations, as described in sections 6.7.2.5 and 6.8.2.5.

The second field gives the length of the data value, in bytes. It is represented as a 16-bit unsigned integer. For the first five data types, this is always 4 (i.e., 32 bits). For the character string data types, this is the number of bytes in the string. The size, in bytes, of the property value is thus 4 plus the value of this field.

The third field actually carries the data. It is anywhere from 4 to 65535 bytes, although implementation restrictions may limit the maximum length.

6.5.5. Query Predicates

Operations that inquire about a range of property values need a specification for that range. We call this specification a "query predicate." A query predicate specifies a range using four parameters:

1. The property value type, which must exactly match the type of a candidate property value.
2. An inclusive lower bound on the range of values.
3. An inclusive upper bound on the range of values.
4. A boolean indicating if the complementary range should be used; if this flag is true, then values outside the specified range will match the predicate.

For unordered types, such as atoms and protection values, the lower bound is used to specify a single predicate value, and the upper bound is unused. Thus, one can specify only "equal to" and "not equal to" predicates on unordered types.

A query predicate has six fields. The first is a 16-bit integer specifying the property value type, as in section 6.5.4. The second is a 16-bit field representing the boolean "complement range" flag. The third and fourth fields are unsigned 16-bit integers giving the lengths, respectively, of the lower and upper bounds. The fifth and sixth fields are the values of the lower and upper bounds, respectively. This means that the upper bound value may potentially start on an unaligned boundary:

```
struct QueryPredicate {
        unsigned short  ValueType;
        unsigned short  ComplementRange;
        unsigned short  LowerBoundSize;
        unsigned short  UpperBoundSize;
        /* next two fields are actually variable-length */
        octet           LowerBound[LowerBoundSize];
        octet           UpperBound[UpperBoundSize];
};
```

The total size, in bytes, of a query predicate is 8 plus the sum of the two length fields.

6.5.6. Iterator keys

An iterator key, used to communicate the state of an iteration between client and server, has an internal structure known only to the particular server implementation. The client must treat it simply as an array of bytes. The length of this array is also implementation dependent; server functions exist for the client to use to find the length of an iterator key.

A client is free to make a faithful copy of an iterator key, but must not modify it.

6.6. Notation and conventions for function descriptions

The file and directory service specifications that follow include detailed descriptions of server functions available to the client. These descriptions are given as function declarations; in this section we will briefly discuss the data type notation used in later sections, the "status codes" returned by all operations, and the details of the iterator mechanism.

6.6.1. Data type notation

Server operations are functions mapping a set of input values to a set of output values. Each of these values has a data type specified. For example:

PropIterNext (*FileHandle* : U_INT, *IterKey* : SEQUENCE OF BYTES) ⇒
(*Status* : INT, *IterKey* : SEQUENCE OF BYTES, *Value* : PROPVALUE, *Protection*: PROPPROTECT)

There are three kinds of identifiers used here, each typographically distinct:

- **Function names**: shown in bold italics, for example ***PropIterNext***.
- **Parameter names**: shown in italics, for example *FileHandle*.
- **Type names**: shown in all upper case, for example STRING.

The parameter data type names are:

UJNT	A 32-bit unsigned integer.
STRING	A two-field structure: the first field is a 16-bit unsigned integer, which gives the length of the second field. The second field is any number of ASCII characters, not including nulls.
SEQUENCE OF BYTES	A two-field structure: the first field is a 32-bit unsigned integer, which gives the length of the second field. The second field is any number of bytes, including nulls.
PROPVALUE	A property value as specified in section 6.5.4.
PROPPROTECT	A protection value as specified in section 6.5.2.
PREDICATE	A query predicate as specified in section 6.5.5.

6.6.2. Function status codes

For every function specified in this chapter, the output values include a *Status* value, indicating either that the operation succeeded, or why it failed. Status values are listed in table 6-2. If the *Status* value is not "OK," then the value of the other returned parameters is undefined unless otherwise noted.

1. OK: The operation was successful.
2. **Invalid parameter:** One of the input values was invalid.
3. **Protection** violation: The operation would have violated an access control.
4. **Authentication failure:** User authentication information was not correct.
5. **Not found:** The specified datum was not found.
6. **No such index: A** global query cannot be performed because no index on the specified property exists.
7. **No server resources:** The server lacks sufficient resources to fulfill the request.
8. Handle **invalid:** A temporary file handle is not valid, perhaps because of a server crash.
9. **Lock conflict:** An operation conflicts with an existing lock.
10. **Lock timeout:** The lock associated with the file handle used in this operation has timed out and is broken.
11. **Breaking lock:** Not actually a failure; a *LockQ* operation has succeeded but has broken an existing, timed-out lock.
12. **Derivation conflict:** Specific to the *Crea te Version Like()* operation.
13. **Quota exceeded:** The server is unwilling to allocate further resources to the specific client or user.
14. **Error:** Other error conditions.
15. **Bug:** An implementation bug was detected.
16. **Not Implemented:** The attempted operation was not available in the current implementation.

Table 6-2: Function status codes

6.6.3. Iterators

The servers provide a number of services that involve iteration: for example, iteration over the properties on a property list or the entries in a directory. These iteration operations are all similar in style.

Each iteration operation involves three functions. The first returns the size of the iterator key used with the other two functions, used by clients to allocate temporary storage for the iterator key. The key size function is guaranteed to return the same value each time it is applied to a particular temporary handle; however, if the server is restarted (thus invalidating all handles), the size of an iterator key may change. This allows an implementation to be changed to use a different iterator key size without confusing client processes.

The second function initializes an iterator key, often taking some parameters to constrain the range of the iteration.

The third function returns two things: the value (such as an entry name) associated with the current iterator key, and the successor of the iterator key. Provided that the underlying database is not modified, this operation is idempotent as long as the handle is valid: it always returns the same output values for a given input value. The order in which values are returned from an iteration is implementation-dependent.

```
char *iterKey;
struct {
        int status;
        int size;
} retVall;
struct {
        int status;
        char *key;
} retVal2;
struct {
        int status;
        char *newKey;
        char *entryName;
} retVal3;

retVail - DirlterKeySize(dirHandle);
if (retVail.status !- OK) abort();

iterKey - malloc(retVall.size);

retVal2 - Dirlterlnit(dirHandle);
if (retVal2.status !« OK) abort ();

copy(retVal2.key, iterKey, retVall.size);

while (TRUE) {
    retVal3 - DirIterNext(dirHandle, iterKey);
    if (retVal3.status !» OK)
        break;
    printf("%s\nff, retVal3.entryName);
    copy(retVal3.newKey, iterKey, retVall.size);
}
```

Figure 6-2: Example of the use of an iterator: listing a directory

Figure 6-2 shows how an iterator might be used. The code is a fragment written in the C language; it prints the names of all entries in a directory. This example leaves out a few details related to reclaiming dynamically-allocated memory, but is otherwise accurate.

6.7. File server specification

The file service specification includes:

- A model of multiple-version files. This is the highest-level data abstraction provided by the file server.
- A set of basic operations on files.
- A set of "intrinsic" file properties known to the file server.
- An extension to the set of operations that improves performance by "vectorizing" multiple simpler operations.
- A set of system management operations.

6.7.1. Multiple-version file model

The basic object managed by our file service is a multiple-version file, or "file history." There is a one-to-one correspondence between a file history and a property list group.

A file with *n* extant versions has *n + 1* property lists in its property list group. The names of *n* of the property lists are the version identifiers of the *n* versions. These identifiers are strings of the language specified by the grammar

```
version-id := version-term | version-id'.'version-term
version-term := [0|1|2|3|4|5|6|7|8|9]+
```

In other words, a version-id is a sequence of non-negative integers, separated by periods. For example, "3.4.5" and "14" are legal version-ids. The most significant (left-most) term cannot be zero, except for the distinguished string "0". A property list with this name is always present, and represents the properties of the file history as a whole.

It is possible to have a file history with no extant versions. This is the state of a file history immediately after it is created.

6.7.2. File operations

The basic file operations are separated into the following subgroups:

- **Handle management**: opening, closing, and locking temporary file handles.
- **File management**: creation and deletion of file histories and versions, and iteration over the versions of a file.
- **File data operations**: reading and writing file contents.
- **File property operations**: creating, removing, modifying, reading, and iterating over file properties.
- **Query operations**: searching for file versions that match a predicate on one of their properties.

In all of the function descriptions that follow, the phrase "if the specified access is allowed to the user by the appropriate access matrix" is implied. Table 6-1 on page 64 indicates the access rights that apply to particular operations.

6.7.2.1. Handle management

Most file server operations take a *FileHandle* as an input parameter. The *FileHandle* encodes authentication information, and in most case refers to a specific file or version.

Open (*UserName* : STRING, *Password* : STRING, *FileUID* : STRING, *VersionSpec* : STRING) ⇒ (*Status* : U_INT, *FileHandle* : U_INT)

This operation creates a *FileHandle* for an existing file or version. The *UserName* and *Password* are used to authenticate the client; further operations on this *FileHandle* will be checked against the identity of the client to control access.

The *FileUID* and *VersionSpec* together denote a particular version. If *VersionSpec* is "0", then the file history is meant, rather than any specific version, and file data operations (***write***(), for example) cannot be applied to the handle.

Close (*FileHandle* : U_INT) ⇒ (*Status* : U_INT)

The *FileHandle* is invalidated, and the server releases the associated resources. Implicitly releases a lock on the handle, if any.

Lock (*FileHandle* : U_INT, *LockMode* : U_INT) ⇒ (*Status* : U_INT)

Acquires a lock of the specified *LockMode* on the file or version denoted by *FileHandle*. *LockMode* can be one of the following integer values:

1. File content lock
2. File version lock
3. File history lock

One should not acquire a file version lock on a handle open for version "0" of a file, since this could cause confusing behavior in conjunction with the inheritance features of ***GetProp***(). Instead, lock the entire file history.

Lock() returns "OK" if the lock is successfully acquired. It may also return "Breaking lock" if the lock is successfully acquired by breaking an existing timed-out lock; in this case, the client need to verify the state of the file. If the lock is denied because of a conflicting lock, *Status* is "Lock conflict".

A lock may be *upgraded* by calling ***Lock***() with a higher *LockMode*; if the new mode does not conflict with other locks, the lower mode lock is released and the higher mode lock is acquired. This is done indivisibly. If the new mode would conflict, the existing lock is maintained and the returned *Status* is "Lock conflict".

UnLock (*FileHandle* : U_INT) ⇒ (*Status* : U_INT)
Releases any lock held on the file, version, or contents via this *FileHandle*. Must also be used to clear a "broken lock" condition.

The ***Open***() operation does not have a *Mode* parameter, unlike the usual arrangement where a client requests read or write access when opening a file. Instead, protection is checked on every operation, thus allowing revocation of access to an open file.

6.7.2.2. File management

These operations create and destroy file histories and file versions, and support iteration over the versions of a file.

CreateFile (*UserName* : STRING, *GroupName* : STRING, *Password* : STRING) ⇒ (*Status* : U_INT, *FileUID* : STRING)
If the authentication information is valid, a new file is created, owned by *UserName*. The file is empty except for a file history property list, with default initial values for the intrinsic properties. No access controls apply, although the server may impose quotas that restrict the number of files a user may own.

DeleteFile (*UserName* : STRING, *Password* : STRING, *FileUID* : STRING) ⇒ (*Status* : U_INT)
If the authentication information is valid, all storage associated with the file is marked for reclamation, and the *FileUID* is permanently invalidated. However, if any *FileHandles* exist, the file does not actually cease to exist until the last one is closed.

CreateVersion (*FileHandle* : U_INT, *VersionSpec* : STRING) ⇒ (*Status* : U_INT)
If a version identified by *VersionSpec* does not already exist, it is created with empty contents and a property list with default initial values for the intrinsic properties. A version is owned by the owner of the file, not necessarily the user who issues the ***CreateVersion***() operation.

DeleteVersion (*FileHandle* : U_INT, *VersionSpec* : STRING) ⇒ (*Status* : U_INT)
The specified file version (and associated property list) is deleted. However, VersionSpec may not be "0", since the file history property list must always exist.

VersionIterKeySize (*FileHandle* : U_INT) ⇒ (*Status* : U_INT, *Size* : U_INT)
Returns the size, in bytes, of an iterator key for iterating over the versions of a file.

VersionIterInit (*FileHandle* : U_INT) ⇒ (*Status* : U_INT, *IterKey* : SEQUENCE OF BYTES)
Returns the initial value in a sequence of iterator key values for iterating over the versions of a file history.

VersionIterNext (*FileHandle* : U_INT, *IterKey* : SEQUENCE OF BYTES) ⇒
(*Status* : U_INT, *NewIterKey* : SEQUENCE OF BYTES, *VersionSpec* : STRING)
Given an iterator key value, returns the successor value in *NewIterKey* and the corresponding version specifier in *VersionSpec*.

6.7.2.3. File data operations

The operations in this group are similar to the data transfer operations used in many operating systems. Because they are intended for use in a distributed system we want to minimize client state stored at the server, so they take an explicit *ByteOffset* parameter. Some files systems maintain an implicit offset on behalf of the client, but we require the client to maintain the offset pointer.

Write (*FileHandle* : U_INT, *ByteOffset* : U_INT, *Buffer* : SEQUENCE OF BYTES) ⇒ (*Status* : U_INT)
The specified *Buffer* is written to the file, starting at offset *ByteOffset* from the beginning of the file. If *ByteOffset* is greater than the current length of the file, a "hole" is created; the values of the bytes in a hole is undefined.

If the protection of the file allows the user to append only, then *ByieOffset* must be equal to the current "Length" property of the file. When attempting to append data to a file version, it is a good idea to lock the version before reading the "Length" property, and release the lock after performing the *WriteQ*.

Read *(FileHandle*: U INT, *ByteOffset*: U_INT, *WantedLength* : U_INT) =•
"(*Status*: UJNT, *Buffer*: SEQUENCE OF BYTES)

Returns the *WantedLength* bytes starting at offset *ByteOffset* from the beginning of the file. If *ByteOffset* is greater than the length of the file, an "Invalid Parameter" status is returned. If the length of the file is less than *(ByteOffset + WantedLength)* bytes, then the returned *Buffer* will be shorter than *WantedLength.*

FlushBuffers (FileHandle: UJNT) => *(Status*: UJNT)

When this function completes successfully, the server guarantees that all permanent data related to the file version on which the *FileHandle* is open is stored in non-volatile storage. This means that a server crash after a *FlushBuffersQ* operation and before any operation that modifies file data or properties will not affect the integrity of the file. Server crashes at other times may result in lost data.

Truncate *(FileHandle*: UJNT, *ByteOffset*: UJNT) => *(Status*: UJNT)

Sets the lenglh of the file to *ByteOffset,* provided that *ByteOffset* is not greater than the length of the file. This function requires WRITE access to the file contents.

6.7.2A File property operations

These operations provide access to the properties on a particular property list, corresponding either to a specific version or to the file as a whole. Since the *OpenQ* operation takes a version specifier as well as a *FileUID,* a *FileHandle* uniquely specifies a single property list.

PutProp *(FileHandle*: U_INT, *PropName*: STRING, *Value*: PROPVALUE, *Protection:* PROPPROTECT) =*
(Status: UJNT)

If a property with name *PropName* exists, its value is changed to *Value,* if it does not exist, it is created with the specified *Value* and *Protection.*

***PutPropQ* cannot change the type of an existing property; the *Type* field of the *Value* parameter must match the type of an existing property. The *Protection* of an existing property can be changed only if it grants APPEND access to the client.**

GetProp (FileHandle: U_INT, *PropName*: STRING) =>
***(Status*: UJNT, *Value:* PROPVALUE, *Protection:* PROPPROTECT)**

Returns the *Value* and *Protection* of the property named *PropName,* if it exists or can be inherited from the file history. If it does not exist, the returned status is "Not Found".

RemProp (FileHandle: UJNT, *PropName :* STRING) => *(Status*: UJNT)

Removes the property named *PropName,* if it exists. Otherwise, returns "Not Found".

PropIterKeySize *(FileHandle*: UJNT) => *(Status:* UJNT, *Size:* UJNT)

Returns the size, in bytes, of an iterator key for iterating over the properties on a property list.

PropIterInit *(FileHandle:* UJNT) => *(Status:* UJNT, *IterKey*: SEQUENCE OF BYTES)

Returns the initial value in a sequence of iterator key values for iterating over the properties on the property list for which *FileHandle* is open.

PropIterNext *(FileHandle:* U_INT, *IterKey:* SEQUENCE OF BYTES) =>
***(Status:* U_INT, *NewIterKey*: SEQUENCE OF BYTES, *PropName*: STRING,**
***Value:* PROPVALUE, *Protection:* PROPPROTECT)**

Given an iterator key value, returns the successor value in *NewIterKey* and the corresponding property name, value, and protection in *PropName, Value,* and *Protection* respectively.

Inheritance slightly complicates the picture for ***GetPropQ.*** A property value is *inherited* from the file history (version "0") property list if it exists there but is not on the list for the specified version. For example, if the property named "Type" appears on the version "0" list but not on the list for version "3", then a *GetPropQ* for that property applied to a file handle for version "3" will return the value from the file history list.

The usual case is that inheritance is desired, but sometimes a client needs to know if a property actually appears on the version property list, especially when preparing to modify the property. For this case, a special version of ***GetPropQ*** exists that does not do inheritance:

GetPropNoInherit *(FileHandle:* U_INT, *PropName:* STRING) =>
***(Status:* UJNT, *Value:* PROPVALUE, *Protection:* PROPPROTECp**

Returns the *Value* and *Protection* of the property named *PropName,* if it exists. Otherwise, returns "Not Found".

If a *GetPropQ* would inherit a value from a file history list that is locked on a different file

handle, the operation will fail with a "Lock conflict" *Status*. A client should be prepared to handle this error, even though it may hold a lock on the specific version property list.

No function besides ***GetProp***() performs inheritance.

6.7.2.5. Query operations

The file service supports two kinds of "query" operations, differing in the breadth of their search. *File queries* search the versions (property lists) of a specific file; *global queries* search the property lists of all the files stored by a single file server. Otherwise, they are essentially the same: each is an iterator that takes a property name and *query predicate* and returns the set of property values that meet the predicate, as well as identification of the property list on which each property is found.

A single property name can be bound on different property lists to values of several types. A query operation only tests values of a single type against its query predicate, and thus ignores values of other types. There is one exception to this rule: if the type field of the query predicate is "No Type" (that is, zero), then the other fields of the predicate are ignored, and every property with the given property name is returned. This allows a client to discover all occurrences of a particular property name.

File queries are performed using the functions:

FileQueryKeySize (*FileHandle* : U_INT) ⇒ (*Status* : U_INT, *Size* : U_INT)
Returns the size, in bytes, of an iterator key for performing a query over the property lists of a single file.

FileQueryInit (*FileHandle* : U_INT, *PropName* : STRING, *QuerySpec* : PREDICATE) ⇒
(*Status* : U_INT, *IterKey* : SEQUENCE OF BYTES)
Returns the initial value in a sequence of iterator key values denoting the properties in the file history, with name *PropName*, whose values meet the predicate *QuerySpec*.
FileHandle can be for any version of the file in question, including the "0" version; the particular version does not affect the result of the query.

FileQueryNext (*FileHandle* : U_INT, *IterKey* : SEQUENCE OF BYTES) ⇒
(*Status* : U_INT, *NewIterKey* : SEQUENCE OF BYTES, *VersionSpec* : STRING,
Value : PROPVALUE)
Given an iterator key value, returns the successor value in *NewIterKey* and the corresponding version identifier and property value in *VersionSpec* and *Value*; the *Value* and name of the property match the query predicate and name specified in ***FileQueryInit***().

Global queries are performed using the functions:

GlobalQueryKeySize (*FileHandle* : U_INT) ⇒ (*Status* : U_INT, *Size* : U_INT)
Returns the size, in bytes, of an iterator key for performing a query over all the property lists stored by the file server.

GlobalQueryInit (*FileHandle* : U_INT, *PropName* : STRING, *QuerySpec* : PREDICATE) ⇒
(*Status* : U_INT, *IterKey* : SEQUENCE OF BYTES)
Returns the initial value in a sequence of iterator key values denoting the properties stored by the file server, with name *PropName*, whose values meet the predicate *QuerySpec*.
FileHandle can be for any version, including the "0" version, of any file stored by the server; the particular file or version does not affect the result of the query. For global queries, *FileHandle* is only used to denote authentication information.
This function may fail with a *Status* of "No Index" if no index exists to support the query.

GlobalQueryNext (*FileHandle* : U_INT, *IterKey* : SEQUENCE OF BYTES) ⇒
(*Status* : U_INT, *NewIterKey* : SEQUENCE OF BYTES, *FileUID* : STRING,
VersionSpec : STRING, *Value* : PROPVALUE)
Given an iterator key value, returns the successor value in *NewIterKey* and the corresponding file and version identifiers and property value in *FileUID*, *VersionSpec* and *Value*, respectively; the *Value* and name of the property match the query predicate and name specified in ***GlobalQueryInit***().
GlobalQueryNext() skips any instances of the property to which the client does not have READ access.

Although ***PropIterNext***() returns the property protection, neither ***FileQueryNext***() nor ***GlobalQueryNext***() does so. This reflects an educated guess as to the relative usefulness of the protection information in the situations where the three iterative operations are used. In any case, ***GetProp***() can be used to retrieve the protection.

Query operations do not perform implicit locks. Instead, the client must either verify the consistency of the query results, or acquire a lock on the appropriate file(s) before doing the query. Global queries, especially, should not be relied upon to return a consistent view of the file system. It is unrealistic to lock the entire file system during a global query.

6.7.3. Intrinsic properties

The file service "understands" a small set of intrinsic properties of files. These properties are implicitly read or written during some operations.

In the description that follows, we give for each intrinsic property its name; its value data type; whether it is a property of a version, a file history, or both; whether it can be modified with ***PutProp***(); its meaning; and a list of the operations that implicitly read or write it.

The protection of an intrinsic property is treated slightly differently from that of other properties. An intrinsic property can never be removed by ***RemProp***(); "unmodifiable" intrinsic properties, such as "Length", cannot have their values changed explicitly by ***PutProp***(). However, we wish to preserve the ability for suitably authorized users, such as the "owner," to change the protection of the property, with ***PutProp***(), so as to grant or deny read access to other users. Therefore, we disallow the setting, via ***PutProp***(), of the DELETE access right for any intrinsic property, and of the WRITE access right for unmodifiable intrinsic properties. The APPEND (i.e., change protection) and READ access rights can be modified.

Unless otherwise mentioned, the default protection for newly created unmodifiable intrinsic properties is "U:RA G:R O:R E:", and "U:RWA G:R O:R E:" for modifiable ones.

Length (Integer, version)
- The length of the file, in bytes. The initial value is zero.
- Modified by: ***Write***(), ***Truncate***().
- Read by: ***Write***() and ***Read***(), to locate end-of-file.

Owner (Atom, history, modifiable)
- The user that owns the file history. The initial value is set from a parameter of the ***CreateFile***() operation.
- Modified by: ***CreateFile***().
- Read by: all but ***Open***() and ***Close***(), as part of verifying access rights.

Group (Atom, history, modifiable)
- The user that owns the file history. The initial value is set from a parameter of the ***CreateFile***() operation.
- Modified by: ***CreateFile***().
- Read by: all but ***Open***() and ***Close***() operations, as part of verifying access rights.

Create_Time (Time, history and version)
- The time when the history or version was created.
- Modified by: ***CreateFile***() or ***CreateVersion***(), as appropriate.
- Read by: none, but might be used in archiving or system maintenance.

Modify_Time (Time, version, modifiable)
- The time when the contents of the version were last modified.
- Modified by: ***Write***().
- Read by: none, but might be used in archiving or system maintenance.

Read_Time (Time, version, modifiable)
- The time when the contents of the version were last read.
- Modified by: ***Read***().
- Read by: none, but might be used in archiving or system maintenance.

Last_Writer (Atom, version)
- The user that last wrote the contents of the version.

- Modified by: ***Write***().
- Read by: none.

Protection (Protection, history and version, modifiable)

- Controls the creation and listing of versions of a file history, and deletion of the entire history. Controls access to the contents of a file version. The initial value is "U:RWA, G:R, O:, E:".
- Modified by: none.
- Read by: For file histories, ***DeleteFile***(), ***CreateVersion***(), ***VersionIterInit***(), and ***VersionIterNext***(). For file versions, ***Write***() and ***Read***().

List_Protection (Protection, history and version, modifiable)

- Controls access to the property list on which it is found, including listing and creating new properties, and deleting the list itself. The initial value is "U:RDA, G:R, O:, E:".
- Modified by: none.
- Read by: ***PropIterInit***(), ***PropIterNext***(), ***PutProp***(), ***DeleteVersion***(), and ***DeleteFile***().

Lock_State (Boolean, version)

- Indicates if the version contents are locked.
- Modified by: ***Lock***(), ***UnLock***().
- Read by: All operations.

Lock_Timeout (Time, version)

- If the version contents currently locked, indicates when the lock will time out. Otherwise, the value is undefined.
- Modified by: ***Lock***(), ***UnLock***(), and updated upon any activity on the locked file handle.
- Read by: ***Lock***(), to allow breaking of lock.

Lock_Holder (String, version)

- If the version contents currently locked, indicates the user that holds the lock. Otherwise, the value is undefined.
- Modified by: ***Lock***() and ***UnLock***()
- Read by: None.

In addition to the intrinsic properties described above, which are part of the external specification of the file server, a file server implementation will probably store other intrinsic properties necessary for its operation. Among these "hidden" intrinsic properties might be a pointer to the contents of a file version, "dirty bits" used to implement incremental backup, profiling information to support automatic migration, etc.

6.7.4. Composite operations

The operations described in section 6.7.2 are complete, in the sense that they provide a client access to all the functions of the file server. However, they may not be acceptably efficient, not because of the cost of the database manipulations at the server, but because of the communication cost of repeated remote operations.

The basic operations were designed so that the amount of information passed for any single function is bounded, at least by the client[5]. This policy allows simple storage management at the client, and potentially a simpler RPC implementation, but requires that some operations that logically could be indivisible to be carried out as a sequence of much more primitive steps. The operations defined in this section provide indivisible alternatives to such sequences, and hence dramatically reduce communication costs. (Specific performance improvements are discussed in section 8.3.3.)

These composite operations gain efficiency in two ways: by combining a large number of

---

[5]The bound is somewhat fuzzy, but is sufficient so long as strings used for property names, version names, and property values are not "unreasonably long."

simpler operations, they reduce the overhead of remote operation invocation, or they avoid unnecessary data transfer by performing at the server those operations that logically take place at the server, such as file copying. An implementation has the option of not providing these operations; they could instead be provided in library routines at the client, albeit with considerable performance loss.

6.7.4.1. Vector data types

"Vector" operations reduce communication overhead by returning the results of an implicit iteration in one or more variable-length vectors. There are two vector data types, one carrying atoms and the other carrying property values.

A vector has two or more fields. The first field is a 32-bit unsigned integer giving the total length of the vector, in bytes. (This is inclusive of the two header fields.) The second field is a 32-bit unsigned integer giving *N*, the number of elements of the vector. The *N* element fields immediately follow: STRINGS for the STRINGJ/ECTOR data type and PROPVALUEs for the PROPJ/ECTOR data type. Both STRING and PROPJVALUE data types are variable-length, but since each encodes its own length it is possible to locate the boundaries between vector elements.

Vector values can be arbitrarily large, so a client must be prepared either to allocate substantial dynamic memory, or detect overflow and fall back on non-vector operations when the vector is too large. Alternatively, by violating the layering of RPC, a client could process vector elements incrementally as they arrive in the communication stream.

Since a server never receives, but only sends, vectors, its memory allocation requirements are less stringent. A server can either return a "No server resources" status, or construct the vector incrementally if it does not fit in memory[6].

6.7.4.2. Vector operations

These operations return a vector of results. Since the "elements" of the results might be pairs or triples of values, rather than returning heterogeneous records these operations return parallel arrays of simple values. The *nth* element of the result is composed of the *nth* component of the first vector, the nth component of the second vector, etc. The order in which result elements appear is otherwise implementation-dependent.

***VersionVector(FileHandle*: UJNT) => *(Status*: UJNT, *Versions*: STRING_VECTOR)**
Returns in *Versions* a vector of the version identifiers for the file history for which *FileHandle* is open.

***PropVector(FileHandle*: UJNT) =* *(Status*: UJNT, *PropNames*: STRINGJ/ECTOR, *PropValues*: PROP_VECTOR)**
Returns two equai-iength vectors, giving in *PropNames* the property names for the file version for which *FileHandle* is open, and in corresponding elements of *PropValues* the values of those properties.

***FileOueryVector(FileHandle*: U_INT, *PropName*: STRING, *QuerySpec*: PREDICATE) =>**
***(Status*: UJNT, *Versions*: STRING_VECTOR, *Properties*: PROP_VECTOR)**
For the file history for which *FileHandle* is op"en, returns two equal-length vectors, giving in *Versions* the version identifiers of the property lists on which a property named *PropName* matched the predicate *QuerySpec,* and in corresponding elements of *PropValues* the values of those properties.

***GlobalQueryVector(FileHandle*: U_INT, *PropName*: STRING, *QuerySpec*: PREDICATE) =»**
***(Status* : U INT, *FileUIDs*: STRING_VECTOR, *Versions*: STRING_VECTOR,**
Properf/es7PROP_VECTOR)
Returns three equal-length vectors, giving in *FileUIDs* and *Versions* the file and version identifiers of the property lists on which a property named *PropName* matched the predicate *QuerySpec,* and in corresponding elements of *PropValues* the values of those properties.

---

incremental construction may require a two-pass algorithm, since the size and length of the vector must precede the data.

If more than one version of a given file has a matching property, the corresponding file identifier will appear more than once in *FileUIDs.*

6.7.4.3. Short-circuit operations

Under a strict layering regime, some operations involve transferring significant amounts of data from server to client and then back to the server. "Short-circuit" operations [Nowi85] are those that keep the data at the server, significantly reducing the communication costs.

Renaming a file version (that is, changing the version identifier of a property list) is one such operation. To do this using the basic operations of section 6.7.2 would involve creating a new version with the new name, transferring the version contents and properties from the old version to the new version, and finally removing the old version. Not only does this require a lot of unnecessary communication, but it also fails to preserve the values of some intrinsic properties (such as CreateJTime). The *VerswnRenameQ* function provides a simple and efficient alternative.

VersionRename (FileHandle: UJNT, *NewVersionSpec*: STRING) => *(Status*: UJNT)

Changes the version identifier of the version for which *FileHandle* is open to *NewVersionSpec.*

The user must have both APPEND (i.e., create version) access to the file history, and DELETE access to the version property list, to perform this function. Neither version identifier can be "0".

Copying an entire file or a single version, if done by a client, requires the entire contents of the file to be communicated twice. The *CopyFileQ* and *CopyVersionQ* functions short-circuit this data transfer; they also allow a file server implementation to use copy-on-write techniques to improve run-time and space efficiency.

CopyFile (FileHandle: UJNT) => *(Status* : UJNT, *FileUID*: STRING)

Creates an exact copy, except for "Creation_Time" and possibly "Owner" and "Group" properties, of the file history for which *FileHandle* is open, and returns the *FifeUID* of the copy.

The user must have READ access to all properties, property lists, contents, and the file history itself.

CopyVersion (SrcFileHandle: UJNT, *DstFileHandle*: UJNT, *DstVersionSpec*: STRING) => *(Status* : UJNT)

Creates an exact copy, except for the "CreatfonjIme" property, of the file version for which"SrcF//eHanc//e is open. The copy is created as version *DstVersionSpec* of the file history for which *DstFileHandle* is open, unless that version already exists.

The user must have READ access to the properties, contents, and property list of the source version, and APPEND (i.e., create version) access to the destination file history.

Note that the two copy functions do no overwrite existing files or versions; they only create new versions. To replace an existing version, one can use *CopyVersionQ* together with *VerswnRenameQ.*

When a client wants to create a new version of an existing file, it must provide an explicit *VersionSpec* to the *CreateVersionQ* function. Determining what version identifier to use requires two pieces of information: which "branch" of the version tree the new version should be on, and what the highest version number on that branch is. A client wishing to read the latest version on a branch has a similar problem. Given the branch location, a library routine could iterate over the versions of the file to find the appropriate answer. However, the *HighestVersionQ* function may be provided by the server to avoid the unnecessary data transfer of the iteration.

HighestVersion (FileHandle: UJNT, *OldVersionSpec*: STRING) => *(Status*: UJNT, *HighestSuccessor*: STRING)

Given *OldVersionSpec,* returns the highest successor version on the same branch. For example, if *OldVersionSpec* is "3.1" then *HighestSuccessor* might be "3.5" but would never be "4[M] or "3.5.2".

HighestVersionQ does not correct for "missing" nodes in the version tree. For example, if versions 3.4 and 3.5.1 exist but 3.5 does not, *HighestSuccessor* will be 3.5, not 3.6.

In most cases, versions will be single integers, and the dominant use of *HighestVersionQ* will be to find the highest successor of version "0[>J].

6.7.4.4. Derivation operations

In section 4.4, we discussed the problem of preserving file properties across derivations. The file service provides two operations to assist in this:

CreateFileLike (*UserName* : STRING, *GroupName* : STRING, *Password* : STRING, *OldFileHandle* : U_INT) ⇒ (*Status* : U_INT, *FileUID* : STRING)
Like ***CreateFile*** except that the initial file-history property list is a copy of that of the file history referred to by *OldFileHandle*. The "Owner" and "Group" properties are not copied, but are set from the *UserName* and *GroupName* parameters.

CreateVersionLike (*FileHandle* : U_INT, *VersionSpec* : STRING, *OldFileHandle* : U_INT) ⇒ (*Status* : U_INT)
Like ***CreateVersion*** except that the initial version property list is a copy of that of the file version referred to by *OldFileHandle*.
This operation fails with a *Status* of "Derivation conflict" if any property on the new version's property list would override a property from its file history property list.

6.7.5. System management operations

It is an implementation option to automatically create an index, for use in answering global queries, for every property name. If indices are not automatically created, then the system manager or possibly other users may choose to create them for specific properties. In either case, the following pair of operations controls the existence of inverted indices.

CreateIndex (*UserName* : STRING, *Password* : STRING, *PropName* : STRING) ⇒ (*Status* : U_INT)
Causes creation of an inverted index on property *PropName*, owned by *UserName*.
This operation may take some time to complete, since it requires the system to check all extant files while constructing the index.
An implementation may restrict this operation to a specified group of users, since it commits system-wide run-time and storage resources.

DeleteIndex (*UserName* : STRING, *Password* : STRING, *PropName* : STRING) ⇒ (*Status* : U_INT)
The inverted index on property *PropName*, if it exists, is deleted, if the creator of the index is *UserName*.

The following function is useful when creating or rebuilding an index:

FileIndexify (*FileHandle* : U_INT, *PropName* : STRING) ⇒ (*Status* : U_INT, *Added* : U_INT)
Adds the *PropName* property, for each version in the file history for which *FileHandle* is open, to the index on *PropName*. Returns the number of values added.

A system manager may need to iterate over all the files stored by a file server, without relying on a directory service. For example, this may be done in conjunction with ***FileIndexify***() to rebuild an index. The file server supports both iterative and vectored versions of this operation:

FileIDIterKeySize (*FileHandle* : U_INT) ⇒ (*Status* : U_INT, *Size* : U_INT)
Returns size, in bytes, of an iterator key for iterating over the files stored by the file server.

FileIDIterInit (*FileHandle* : U_INT) ⇒ (*Status* : U_INT, *IterKey* : SEQUENCE OF BYTES)
Returns the initial value in a sequence of iterator key values for iterating over the files stored by the file server.

FileIDIterNext (*FileHandle* : U_INT, *IterKey* : SEQUENCE OF BYTES) ⇒ (*Status* : U_INT, *NewIterKey* : SEQUENCE OF BYTES, *FileUID* : STRING)
Given an iterator key value, returns the successor value in *NewIterKey* and the corresponding file identifier in *FileUID*.

FileIDVector (*FileHandle* : U_INT) ⇒ (*Status* : U_INT, *FileUIDs* : STRING_VECTOR)
Returns a vector containing the file identifiers for every file stored by the file server.

6.8. Directory server specification

The directory service specification closely parallels the file service specification. It includes:

1. A model of directories, the highest-level data abstraction provided by the directory server, and of directory entries.
2. A set of basic operations on directories.
3. A set of "intrinsic" properties known to the directory server.
4. A set of extended operations to improve performance.
5. A set of system management operations.

6.8.1. Directory model

The directory service manages a set of objects called *directories* or *directory nodes*. A directory node is a set of *directory entries* that refer to objects such as files and directory nodes. A directory graph of arbitrary complexity can be built out of these objects.

More concretely, a directory node is a property list group, and a directory entry is a named property list. The entry refers to an object by properties denoting the object's identifier, type, and server; it may also have other properties.

A directory always has one distinguished entry, named ".", which refers to itself. Thus, one can associate properties with the directory node as a whole by placing them on the "." list.

6.8.2. Directory operations

The basic operations on directories fall into several subgroups:

- **Handle management**: opening, closing, and locking temporary handles.
- **Directory management**: creation and deletion of directory nodes, and iteration over the entries of a node.
- **Entry management**: creation and deletion of directory entries.
- **Directory entry property operations**: creating, removing, modifying, reading, and iterating over directory properties.
- **Query operations**: searching for directory entries that match a predicate on one of their properties.

There is no "Lookup" operation in the basic set; this is because ***Lookup***(), while provided as a composite operation (described in section 6.8.4), can be constructed out of the basic operations.

In all of the function descriptions that follow, the phrase "if the specified access is allowed to the user by the appropriate access matrix" is implied. Table 6-1 on page 64 indicates which access rights apply to which operations.

6.8.2.1. Handle management

Most directory server operations take a *DirHandle* as an input parameter. This differs from those directory systems that are closely integrated with an operating system kernel, but with the directory service as a separate entity, client authentication becomes an issue. The *DirHandle* encodes authentication information, and refers to a specific directory node.

DirOpen (*UserName* : STRING, *Password* : STRING, *DirUID* : STRING) ⇒ (*Status* : U_INT, *DirHandle* : U_INT)
Creates a *DirHandle* for the existing directory denoted by *DirUID*. The *UserName* and *Password* are used to authenticate the client; further operations on this *DirHandle* will be checked against the identity of the client to control access.

Close (*DirHandle* : U_INT) ⇒ (*Status* : U_INT)
The *DirHandle* is invalidated, and the server releases the associated resources. Implicitly releases a lock on the handle, if any.

DirLock (*DirHandle* : U_INT) ⇒ (*Status* : U_INT)
Acquires a lock on the directory specified by *DirHandle*.
DirLock() returns "OK" if the lock is successfully acquired. It may also return "Breaking lock" if the lock is successfully acquired by breaking an existing timed-out lock; in this case, the client should to verify the state of the directory. If the lock is denied because of a conflicting lock, *Status* is "Lock conflict".

UnLock (*DirHandle* : U_INT) ⇒ (*Status* : U_INT)
Releases the lock held on the directory via this *DirHandle*. Must also be used to clear a "broken lock" condition.

6.8.2.2. Directory management

These operations create and destroy directory nodes, and support iteration over the entries in a directory.

CreateDir (*UserName* : STRING, *GroupName* : STRING, *Password* : STRING) ⇒ (*Status* : U_INT, *DirUID* : STRING)
If the authentication information is valid, a new directory is created, owned by *UserName*. The directory is empty except for the "." property list, with default initial values for the intrinsic properties. No access controls apply, although the server may impose quotas that restrict the number of directories a user may own.

DeleteDir (*UserName* : STRING, *Password* : STRING, *DirUID* : STRING) ⇒ (*Status* : U_INT)
If the authentication information is valid, all storage associated with the directory is marked for reclamation, and the *DirUID* is permanently invalidated. However, if any *DirHandles* exist, the directory does not actually cease to exist until the last one is closed.

DirIterKeySize (*DirHandle* : U_INT) ⇒ (*Status* : U_INT, *Size* : U_INT)
Returns the size, in bytes, of an iterator key for iterating over the entries of a directory.

DirIterInit (*DirHandle* : U_INT) ⇒ (*Status* : U_INT, *IterKey* : SEQUENCE OF BYTES)
Returns the initial value in a sequence of iterator key values for iterating over the entries of a directory.

DirIterNext (*DirHandle* : U_INT, *IterKey* : SEQUENCE OF BYTES) ⇒
(*Status* : U_INT, *NewIterKey* : SEQUENCE OF BYTES, *EntryName* : STRING)
Given an iterator key value, returns the successor value in *NewIterKey* and the corresponding entry name in *EntryName*.

6.8.2.3. Entry management

These operations create and delete directory entries.

CreateEntry (*DirHandle* : U_INT, *EntryName* : STRING, *ObjectID* : STRING, *ServerID* : STRING, *EntryType* : STRING) ⇒ (*Status* : U_INT)
If a entry identified by *EntryName* does not already exist, it is created with a property list with default initial values for the intrinsic properties. The "ObjectID", "ServerID", and "EntryType" properties are set as specified. An entry is owned by the owner of the directory, not necessarily the user who issues the ***CreateEntry***() operation.

DeleteEntry (*DirHandle* : U_INT, *EntryName* : STRING) ⇒ (*Status* : U_INT)
The specified entry (property list) is deleted. EntryName may not be ".", since the directory's self-referential property list must always exist.

6.8.2.4. Directory entry property operations

These operations provide access to the properties on an entry's property list. To uniquely specify a property list, these operations take both a *DirHandle* and an *EntryName*.

DirPutProp (*DirHandle* : U_INT, *EntryName* : STRING, *PropName* : STRING, *Value* : PROPVALUE, *Protection*: PROPPROTECT) ⇒ (*Status* : U_INT)
If a property with name *PropName* exists, its value is changed to *Value*. If it does not exist, it is created with the specified *Value* and *Protection*.
DirPutProp() cannot change the type of an existing property; the *Type* field of the *Value* parameter must match the type of an existing property. The *Protection* of an existing property can be changed only if it grants APPEND access to the client.

DirGetProp (*DirHandle* : U_INT, *EntryName* : STRING, *PropName* : STRING) ⇒ (*Status* : U_INT, *Value* : PROPVALUE, *Protection*: PROPPROTECT)
Returns the *Value* and *Protection* of the property named *PropName*, if it exists. Otherwise, returns "Not Found".

DirRemProp (*DirHandle* : U_INT, *EntryName* : STRING, *PropName* : STRING) ⇒ (*Status* : U_INT)
Removes the property named *PropName*, if it exists. Otherwise, returns "Not Found".

DirPropIterKeySize (*DirHandle* : U_INT) ⇒ (*Status* : U_INT, *Size* : U_INT)
Returns the size, in bytes, of an iterator key for iterating over the properties on a property list.

DirPropIterInit (*DirHandle* : U_INT, *EntryName* : STRING) ⇒ (*Status* : U_INT, *IterKey* : SEQUENCE OF BYTES)
Returns the initial value in a sequence of iterator key values for iterating over the properties on the property list named *EntryName* of the directory for which *DirHandle* is open.

DirPropIterNext (*DirHandle* : U_INT, *IterKey* : SEQUENCE OF BYTES) ⇒ (*Status* : U_INT, *NewIterKey* : SEQUENCE OF BYTES, *PropName* : STRING, *Value* : PROPVALUE, *Protection*: PROPPROTECT)
Given an iterator key value, returns the successor value in *NewIterKey* and the corresponding property name, value, and protection in *PropName*, *Value*, and *Protection* respectively.

6.8.2.5. Query operations

The directory service supports both *directory queries*, which search the entries (property lists) of a specific directory, and *global queries*, which search the property lists of all the directories stored by a single file server. Otherwise, they are essentially the same: each is an iterator that takes a property name and "query predicate" and returns the set of property values that meet the predicate, as well as identification of the property list on which each property is found.

A single property name can be bound on different property lists to values of several types. A query operation only tests values of a single type against its query predicate, and ignores values of other types. There is one exception to this rule: if the type field of the query predicate is "No Type" (that is, zero), then the other fields of the predicate are ignored, and every property with the given property name is treated as a match. This allows a client to discover all occurrences of a particular property name.

Directory queries are performed using the functions:

DirQueryKeySize (*DirHandle* : U_INT) ⇒ (*Status* : U_INT, *Size* : U_INT)
Returns the size, in bytes, of an iterator key for performing a query over the entries of a single directory.

DirQueryInit (*DirHandle* : U_INT, *PropName* : STRING, *QuerySpec* : PREDICATE) ⇒ (*Status* : U_INT, *IterKey* : SEQUENCE OF BYTES)
Returns the initial value in a sequence of iterator key values denoting the properties in the directory, with name *PropName*, whose values meet the predicate *QuerySpec*.

DirQueryNext (*DirHandle* : U_INT, *IterKey* : SEQUENCE OF BYTES) ⇒ (*Status* : U_INT, *NewIterKey* : SEQUENCE OF BYTES, *EntryName* : STRING, *Value* : PROPVALUE)
Given an iterator key value, returns the successor value in *NewIterKey* and the corresponding entry name and property value in *EntryName* and *Value*; the *Value* and name of the property match the query predicate and name specified in ***DirQueryInit***().

Global queries are performed using functions essentially identical to those used in the file server:

GlobalOueryKeySize (DirHandle: UJNT) =* *(Status*: UJNT, *Size*: UJNT)

Returns the size, in bytes, of an iterator key for performing a query over ail the property lists stored by the directory server.

GlobalQuerylnit (DirHandle : U_INT, *PropName*: STRING, *QuerySpec*: PREDICATE) =»
(Status : UJNT, *IterKey*: SEQUENCE OF BYTES)

Returns the initial value in a sequence of iterator key values denoting the properties stored by the directory server, with name *PropName,* whose values meet the predicate *QuerySpec.*

DirHandle can be for any directory stored by the server; For global queries, *DirHandle* is only used to denote authentication information.

This function may fail with a *Status* of "No Index" if no index exists to support the query.

GlobalQueryNext (DirHandle : UJNT, *IterKey*: SEQUENCE OF BYTES) =>
(Status: UJNT, *NewlterKey*: SEQUENCE OF BYTES, *DirUID*: STRING,
EntryName: STRING, *Value* : PROPVALUE)

Given an iterator key value, returns the successor value in *NewlterKey* and the corresponding directory identifier, entry name, and property value in *DirUID, EntryName* and *Value,* respectively; the *Value* and name of the property match the query predicate and name specified in *GlobalQuerylniH).*

GlobalQueryNextQ skips any instances of the property to which the client does not have READ access.

6.8.3. Intrinsic properties

The directory service "understands" a small set of intrinsic properties of directories and directory entries. These properties are implicitly read or written during some operations.

For each intrinsic directory property, we give its name, its value data type, whether it is a property of a directory (found on the "." entry) or of an entry, whether it can be modified with *PutPropQ,* its meaning, and the operations that implicitly read or write it.

As with intrinsic file properties, an intrinsic directory property cannot have the DELETE access right set, and an unmodifiable intrinsic property cannot have the WRITE access right set. The default protection for newly created unmodifiable intrinsic properties is "U:RA G:R O:R E:'\ and "U:RWA G:R O:R E:" for modifiable ones.

The first two properties together specify the object to which the entry refers. An object is located not only by its *ObjectID,* but by a *ServerlD* for the server that manages it. Both kinds of identifiers are atoms, so that arbitrarily complex identifiers can be used.

ObjectID (Atom, entry, modifiable)

- Identifies the object to which the entry refers.
- Modified by: *CreateEntry().*
- Read by: none.

ServerlD (Atom, entry, modifiable)

- Identifies the server that manages the object.
- Modified by: *Create En try().*
- Read by: none.

All entries also have an *entry type,* indicating what kind of object is referred to. This is used by some of the composite operations as a guide to interpreting the entry.

EntryType (Atom, entry, modifiable)

- Indicates the type of the object referred to by "ObjectID".
- The directory service "understands" this property only if its value is "Directory"; this allows the service to follow pathnames. Otherwise, the entry is treated as a leaf, and no special interpretation is placed on the *EntryType* property.
- Modified by: *CreateEntryQ, CreateSubDit{).*
- Read by: *GotoDirt), Lookup().*

Other intrinsic properties apply to the entry itself.

Owner (Atom, directory, modifiable)

- The user that owns the directory. The initial value is set from a parameter of the *CreateDirQ* operation.

- Modified by: ***CreateDir***().
- Read by: all but ***Open***() and ***Close***() as part of verifying access rights.

Group (Atom, directory, modifiable)
- The user that owns the directory. The initial value is set from a parameter of the ***CreateDir***() operation.
- Modified by: ***CreateDir***().
- Read by: all but ***Open***() and ***Close***() as part of verifying access rights.

Create_Time (Time, entry)
- The time when the entry was created. (The time of creation for the "." entry is the time of creation of the directory itself.)
- Modified by: ***CreateDir***() or ***CreateEntry***(), as appropriate.
- Read by: none, but might be used in archiving or system maintenance.

Modify_Time (Time, entry, modifiable)
- The time when a property of the entry was last modified.
- Modified by: ***DirPutProp***(), ***DirRemProp***().
- Read by: none, but might be used in archiving or system maintenance.

Read_Time (Time, entry, modifiable)
- The time when a property of the entry was last read.
- Modified by: ***DirGetProp***(), ***DirPropIterNext***(), ***DirQueryNext***().
- Read by: none, but might be used in archiving or system maintenance.

Protection (Protection, directory, modifiable)
- Controls the creation and listing of entries of a directory, and deletion of the directory. The initial value is "U:RWA, G:R, O:, E:".
- Modified by: none.
- Read by: ***DeleteDir***(), ***CreateEntry***(), ***DirIterInit***(), and ***DirIterNext***().

List_Protection (Protection, entry, modifiable)
- Controls access to the property list on which it is found, including listing and creating new properties, and deleting the list itself. The initial value is "U:RDA, G:R, O:, E:".
- Modified by: none.
- Read by: ***DirPropIterInit***(), ***DirPropIterNext***(), ***DirPutProp***(), ***DeleteEntry***(), and ***DeleteDir***().

6.8.4. Composite operations

As with the file service, the directory service provides both vectored composite operations and "short-circuit" composite operations.

6.8.4.1. Vector operations

EntryVector (*DirHandle* : U_INT) ⇒ (*Status* : U_INT, *Entries* : STRING_VECTOR)
Returns in *Entries* a vector of the entry names for the directory for which *DirHandle* is open.

PropVector (*DirHandle* : U_INT, *EntryName* : STRING) ⇒
(*Status* : U_INT, *PropNames* : STRING_VECTOR, *PropValues* : PROP_VECTOR)
Returns two equal-length vectors, giving in *PropNames* the property names for the entry named *EntryName* of the directory for which *DirHandle* is open, and in corresponding elements of *PropValues* the values of those properties.

DirQueryVector (*DirHandle* : U_INT, *PropName* : STRING, *QuerySpec* : PREDICATE) ⇒
(*Status* : U_INT, *Entries* : STRING_VECTOR, *Properties* : PROP_VECTOR)
For the directory for which *DirHandle* is open, returns two equal-length vectors, giving in *Entries* the entry names of the property lists on which a property named *PropName* matched the predicate *QuerySpec*, and in corresponding elements of *PropValues* the values of those properties.

***GlobalQueryVector (Dirhiandle*: U_INT, *PropName*: STRING, *QuerySpec*: PREDICATE) =* ***
(Status: U INT, *DirUIDs*: STRING_VECTOR, *Entries*: STRING_VECTOR, *Properties*:" *PROPJ/ECTOR)***

Returns three equal-length vectors, giving In *DirUIDs* and *Entries* the directory identifiers and entry names of the property lists on which a property named *PropName* matched the predicate *QuerySpec,* and in corresponding elements of *PropValues* the values of those properties.

If more than entry of a given directory has a matching property, the corresponding directory identifier will appear more than once in *DirUIDs.*

6.8.4.2. Short-circuit operations

Short-circuit operations are not as important for the directory service as they are for the file service, since the directory does not deal with such large data as files. However, there are still some functions that can benefit from short-circuiting.

Renaming a directory entry (that is, changing the name of a property list) is one such operation. To do this using the basic operations of section 6.8.2 would involve creating a new entry with the new name, transferring the properties from the old entry to the new entry, and finally removing the old entry. Not only does this require a lot of unnecessary communication, but it also fails to preserve the values of some intrinsic properties (such as CreateJTime). The *RenameEntryQ* function provides a simple and efficient alternative.

***RenameEntry(DirHandle*: UJNT, *OldEntryName*: STRING, *NewEntryName*: STRING) => *(Status*: UJNT)**

Changes the name of the entry named *OldEntryName,* of the directory for which *DirHandle* is open, to *NewEntryName.* Note that one cannot "rename" an entry from one directory into another with this function. The user must have both APPEND (i.e., create entry) access to the directory, and DELETE access to the entry property list, to perform this function. Neither entry identifier can be ".".

Copying a directory entry is often useful so that references to a file (or other object) can be made from two different naming contexts. This copying can be done by a client, but requires the entire contents of the entry to be communicated twice. The *CopyEntryQ* function short-circuits this data transfer.

CopyEntry(SrcDirHandle*: U INT, *SrcEntryName*: STRING *DstDirHandle:* U_INT, *DstEntryName*: STRING) => *(Status". UJHT)

Creates an exact copy, except for the "Creation_Time" property, of the entry named *SrcEntryName* of the directory for which *SrcDirHandle* is open. The copy is created as entry *DstEntryName* of the directory for which *DstDirHandle* is open, unless that entry already exists.

The user must have READ access to the properties and property list of the source entry, and APPEND (i.e., create entry) access to the destination directory.

Note that this copy function does not overwrite existing entries; it only creates new entries. To replace an existing entry, one can use *CopyEntryQ* together with *RenameEntryQ.*

The final group of short-circuit operations implement a somewhat higher-level interface to the directory service. Since these are among the most common operations, even a small speedup is important.

The *CreateSubDirQ* function is used to create a directory node, enter a reference to it in an existing directory, and establish a "parent pointer" from the new directory to the old one. With this, one can create a tree-structured hierarchy of directories.

The *GotoDirQ* and *LookupQ* each interpret a *pathname,* which is a string denoting a chain of directories to follow. *GotoDirQ* returns a new handle for the directory at the tail of the pathname, and is a useful base for a "change working directory" command[7]. *LookupQ,* in

[7]**Note that the directory service has no concept of "working directory" per se. Each open directory handle represents a naming context, which may be used as a current working directory. Unlike traditional systems where there is a single, implicit current context, this system allows any number of active contexts; ••abbreviation*' to an implicit context name must be provided at a higher level.**

contrast, returns the "vital information" about the tail entry of the pathname, and is thus useful as the penultimate operation to opening a file.

Pathnames in the directory system are represented as STRING_VECTORS, allowing higher layers to impose a syntax amenable to typing and printing. We make no distinction between absolute and relative pathnames; pathname-based functions take a directory handle as an explicit context designator, and so are always relative. There is no distinguished "root" to the directory system, except as might be imposed at a higher level.

CreateSubDir (*DirHandle* : U_INT, *EntryName* : STRING) ⇒ (*Status* : U_INT, *DirUID* : STRING)
Creates a new directory, owned by the user who opened *DirHandle*, and returns the new *DirUID*.
Enters the new directory as *EntryName* in the existing "parent" directory for which *DirHandle* is open. The new directory has both a "." (self-referential) entry, and a ".." entry, referring to the parent.

GotoDir (*DirHandle* : U_INT, *Path* : STRING_VECTOR) ⇒ (*Status* : U_INT, *NewDirHandle* : U_INT)
Starting from the directory for which *DirHandle* is open, follows the chain of entries denoted by *Path*. Provided that all the entries on the chain are valid and refer to directories, opens the "tail" directory and returns *NewDirHandle* for it.

Lookup (*DirHandle* : U_INT, *Path* : STRING_VECTOR) ⇒
(*Status* : U_INT, *ObjectID* : STRING, *ServerID* : STRING, *EntryType* : STRING)
Starting from the directory for which *DirHandle* is open, follows the chain of entries denoted by *Path*. Provided that all the entries on the chain are valid and, except for perhaps the last one, refer to directories, returns the *ObjectID*, *ServerID*, and *EntryType* intrinsic properties of the "tail" entry.

Most modern computing environments allow pattern-based specification of file names; for example, in Unix one can type "ls *.c" or "rm chapter?.mss". While the pattern-matching can be done in application code, there are two advantages to doing it at the file server: all applications may take advantage of the same pattern syntax, and only the matching names must be passed across the network.

We provide functions to support file-name patterns. One is similar to ***DirIter***(), except that it returns only those names matching a specified pattern. The other is a vectored form of this operation, analogous to ***EntryVector***(). Both take a *Pattern* parameter, a STRING which specifies a file name using "?" to match exactly one character, and "*" to match any number of characters (including the empty string); all other characters match themselves, with the usual provision that case is insignificant.

DirPatMatchKeySize (*DirHandle* : U_INT) ⇒ (*Status* : U_INT, *Size* : U_INT)
Returns the size, in bytes, of an iterator key for pattern-based iteration over the entries of a directory.

DirPatMatchInit (*DirHandle* : U_INT, *Pattern* : STRING) ⇒ (*Status* : U_INT, *IterKey* : SEQUENCE OF BYTES)
Returns the initial value in a sequence of iterator key values for iterating over the entries of a directory that match the specified *Pattern*.

DirPatMatchNext (*DirHandle* : U_INT, *IterKey* : SEQUENCE OF BYTES) ⇒
(*Status* : U_INT, *NewIterKey* : SEQUENCE OF BYTES, *EntryName* : STRING)
Given an iterator key value, returns the successor value in *NewIterKey* and the corresponding entry name in *EntryName*.

DirPatMatchVector (*DirHandle* : U_INT, *Pattern* : STRING) ⇒ (*Status* : U_INT, *Entries* : STRING_VECTOR)
Returns in *Entries* a vector of the entry names matching *Pattern* for the directory for which *DirHandle* is open.

We do not provide a pattern-matching analogue to ***Lookup***(); this could only be used with patterns that unambiguously specify a single file name, so would be infrequently applicable and perhaps prone to erroneous use.

6.8.5. System management operations

It is an implementation option to automatically create an inverted index, for use in answering global queries, for every property name. If indices are not automatically created, then the system manager or possibly other users can choose to create them for specific properties. In either case, the following pair of operations controls the existence of inverted indices.

***CreateIndex (UserName*: STRING, *Password*: STRING, *PropName* : STRING) =» *(Status* : UJNT)**

Causes creation of an inverted index on property *PropName,* owned by *UserName~.*

This operation may take some time to complete, since it requires the system to check all extant files while constructing the index.

An implementation may restrict this operation to a specified group of users, since it commits system-wide run-time and storage resources.

***Defeteindex(UserName* : STRING, *Password:* STRING, *PropName* : STRING) => *(Status* : UJNT)**

The inverted index on property *PropName,* if it exists, is deleted, if the creator of the index is *UserName.*

The following function is useful when creating or rebuilding an index:

***DirIndexify(DirHandle* : UJNT, *PropName*: STRING) =» *(Status* : UJNT, *Added:* UJNT)**

Adds the *PropName* property, for each entry in the directory for which *DirHandle* is open, to the index on *PropName.* Returns the number of values added.

A system manager may need to iterate over all the files stored by a directory server, without relying on a directory topology that might not contain all the directories. For example, this may be done in conjunction with *DirlndexifyQ* to rebuild an index. The directory server supports both iterative and vectored versions of this operation:

***DirlDIterKeySize (DirHandle :* UJNT) =» *(Status:* UJNT, *Size :* UJNT)**

Returns size, in bytes, of an iterator key for iterating over the directories stored by the directory server.

***DirlDIteHnit (DirHandle :* UJNT) => *(Status* : UJNT, *IterKey*: SEQUENCE OF BYTES)**

Returns the initial value in a sequence of iterator key values for iterating over the directories stored by the directory server.

***DirlDIterNext (DirHandle* : UJNT, *IterKey:* SEQUENCE OF BYTES) =>**
***(Status :* UJNT, *NewIterKey:* SEQUENCE OF BYTES, *DirUID* : STRING)**

Given an iterator key value, returns the successor value in *NewIterKey* and the corresponding directory identifier in *DirUID.*

***DirlDVector (DirHandle :* UJNT) =>**
***(Status :* UJNT, *DirUIDs*: STRING_VECTOR)**

Returns a vector containing the directory identifiers for every directory stored by the directory server.

6.9. Interactions between file and directory services

Users often search directories to find files with specific properties; for example, one might want a list of the Pascal source files in a directory. A simple approach to such a search is to iterate over the entries in the directory, follow each to the file it refers to, ask the file server for the "Pascal-Source-File" property, and list all the files that have that property. This method will get the right answer, but it requires a lot of communication and does not exhibit much locality in database references.

If we "cache" certain properties of a file in the directory entries that refer to it, then we can perform these searches efficiently. The *DirQueryQ* function will provide the answer in one operation, involving only one server, and restricting database access to a small locale. The price for this is the cache-consistency problem.

In a distributed system, with multiple file servers and directory servers, it is extremely expensive to ensure that property values cached in directory entries have the correct value. We would prefer not to require back-pointers from a file to each directory entry, because this would complicate access control, so we would need some other way to find the invalidated cached properties. Even if this could be done efficiently, certain properties (such as "Modified-Time") are so active that cache-maintenance communication might swamp the distributed system.

We therefore do not maintain cache consistency. For some properties, this would be too hard to do; for others, however, it is not necessary: a property value that is known to be constant can be cached without fear of inconsistency. In many cases, the user or application knows, from the meaning of the property, whether it is safe to cache it. A "Pascal-Source-File"

property *probably* will never change; if it is treated as a hint, there is little danger that it will mislead anyone, yet the improvement in search efficiency is significant. Other examples of properties that can be safely cached appear in chapter 9, where we show how to manage bulletin-board messages (which are immutable).

In our design, there is no way to distinguish a cached property, appearing in a directory entry, from a normal property. A naive user might fail to recognize the distinction, and either fail to verify the value of a cached property, or attempt to update it without writing through to the underlying file property. Although our design could be extended to mark cached properties, perhaps through the property-protection mechanism, we do not believe the added complexity is justified.

Chapter 7
Implementation Strategies

In chapter 6 we presented designs for a file system and a directory system. It is central to this thesis that such designs are not only useful in the abstract, but are useful in practice: we must be able to build efficient implementations. In this chapter, we describe several approaches to implementing these designs, and show how good performance can be achieved.

It is seldom possible, or wise, to implement a complete, production-quality large system as part of a dissertation. Since our primary concern was the performance of property operations, we instead built several different versions of a system that closely follows the directory system design of chapter 6. This is a rough subset of an implementation for a file system, and thus eliminated a lot of irrelevant effort It also made it easier to measure the performance of the implementation.

A production-quality system would require the solution of reliability, crash-recovery, and system management problems that, while important, are outside the scope of this thesis. We tried to avoid design choices that would make solutions to these problems impractical, since our performance results would otherwise be suspect.

We start by listing the goals we expect a good implementation to meet. We then describe some principles, gleaned from both successful and unsuccessful attempts at implementing our system, that we believe allow us to meet these goals.

We then describe three distinctly different implementations of the system. Two are (almost) full implementations, in different technologies, of the design of chapter 6, while the other is a demonstration that an existing file system can be given a new interface that provides a subset of that design.

7.1. Implementation Goals

All implementations of everything should be cheap and infallible. We set our sights a little lower. We would like an implementation of a property-based system to meet these goals:

- It must be reliable: The worst thing a system can do is to destroy data. Users grudgingly accept slow systems, but they avoid unreliable ones. The long-term storage facilities must be the most reliable part of a system, because it represents an irreplaceable link with the system's history.

  Reliability ultimately depends on crash-recovery, since all hardware is fallible. Property storage can be made more reliable than file data storage, because the system better understands the structure of properties and can take advantage of this to provide crash-recovery.

- It must be fast: Users care about the speed of the system. Long response time is disconcerting; low overall throughput reduces the effectiveness of the hardware. Useful applications are unavailable if there is no efficient way to implement them.

Response time for access to properties is always critical. Bandwidth is less often important, although when searching over a large domain it becomes so. Most property accesses are reads, not writes, so an implementation should be optimized for read response time.

- **It must use storage efficiently**: On-line storage is usually a limited resource. Extensible property-based systems inherently require more storage space than non-extensible ones, since the latter can encode data more efficiently.

It is not possible to optimize all these goals at once, nor would this be within the scope of a realistic thesis. Because we believe the reliability issues to be orthogonal to the particular problem of property support, we chose to concentrate on performance. Since we had none of the mechanisms necessary to build a highly reliable system[1], the effort involved in doing so would have detracted from more interesting problems.

We stressed run-time performance over storage efficiency. Since the cost per byte for on-line storage seems to be decreasing faster than both the cost per cycle for computing and the access times for on-line storage, it makes sense to trade some storage space for improved response time.

We chose to compare our implementations against the performance of Unix on analogous functions, reasoning that Unix, as a mature and relatively light-weight operating system with an integrated file system, would well represent the performance possible from "traditional" systems. It is also true that Unix provided the only programming environment we had available with both adequate software tools and network facilities, so this might have influenced our choice.

The usual disclaimer applies: although our implementations all run under Unix, they do not depend on facilities peculiar to Unix, and should work as well in any modern environment. Unix idiosyncrasies are responsible for some of the complexities and limitations of the implementations described in this chapter; when the word "Unix" appears in the text, it probably indicates that Unix was getting in the way.

7.2. Principles for good implementations

If our ultimate implementation approaches optimal performance, this is because it was arrived at after no small amount of experimentation and false starts. In this section, we describe our false starts, and try to articulate the principles behind our more successful efforts.

7.2.1. Learning from failure

Our first few attempts at implementing a property-list based system were indeed failures, in that they performed poorly. In part, we can ascribe the poor performance to an excessively simplistic approach; one should always try a simple approach first, since if it works as well, simplicity is inherently preferable to complexity. In this case, unfortunately, it appears that an implementation must be carefully molded to the way in which properties are used, if it is to perform well.

---

[1]These include an accurate model of the ways the system can fail, a communication mechanism that supports replicated servers, and stable storage that provides the basis for a transaction mechanism [Lamp81].

It is also clear that some of the performance problems were due simply to inappropriate implementation choices. It is never a good idea to ignore the basic principles of algorithm and data structure design; one deserves what one gets for operating a hash table at 99% full.

The initial implementations of the directory system design pointed out its inadequacies, and the experience allowed us to improve the design as well as the implementation. For example, initially all property values were stored as character strings. Not only was this costly in both space and run time, but it also prevented us from even thinking about efficient searching for subranges, since there was little prospect of building practical inverted indices. Only after realizing that integers were often more appropriate than character strings was it possible to conceive of the current, multiple-typed, design.

7.2.1.1. First failure

The first implementation maintained a separate database for each property list group. Each database, in turn, consisted of three sub-databases, built out of four separate Unix files. Thus, the overhead of simply opening these files created excessive costs.

One sub-database was used for mapping from strings (property names and property list names) to compact identifiers. This was based on the Unix *dbm* package, which uses an arcane hashing scheme and requires two Unix files to store its data.

Another sub-database was use to store the property value strings. This provided faster access and more compact storage than the *dbm*-based scheme, but could not map strings to integers and so could not be used for list or property names. Both sub-databases were needed because the first one could not map integers to strings; thus, in many cases a single string value would be stored in both sub-databases, with no apparent connection between the values.

The string database was effectively a fixed block-size storage allocator, using a free-list to implement rapid allocation and deallocation. Each block was a small (24 byte) "chunk"; a string would be built out of a linked list of one or more chunks. although most strings would be short enough to require only one chunk.

The "master" sub-database of the per-group database was a B-Tree [Baye72]. B-Tree records included string addresses for the property name and property value. B-Tree keys were composed of a property list name identifier and a property name identifier; the total length of the key was 32 bits. To look up the value of the "owner" property on the list named "editor", we would use the first sub-database to convert "owner" and "editor" into 16-bit integers, then use their concatenation to find the appropriate record in the B-Tree. Finally, we would use the string address in that record to locate the string, in the second sub-database, that represented the property value. Obviously, this involves many potential disk accesses.

The B-Tree records also included fields that connected them into doubly-linked lists; each linked list corresponded to a property list, to allow enumeration of the properties on that property list. If the link address had specified the exact B-Tree disk block for the next record, following the list could have been done quickly. However, this would have required a lot of link updating whenever a B-Tree modification resulted in the motion of one or more records into a different disk block. Therefore, a link was in fact the B-Tree key for the record it referenced; this allowed B-Tree modification without many link updates, but meant that following each link potentially required several disk accesses.

Most of the B-Tree disk traffic was avoided by caching recently-used disk blocks in a layer under the B-Tree implementation; since the B-Trees seldom got very large, the cache hit rate was high and access to existing B-Tree records took essentially constant time. (Our B-Tree implementation, somewhat evolved beyond the state used here, is described in more detail in section 7.3.4.)

We could have improved the performance of this implementation by adding more caches, especially to both string-mapping sub-databases. However, it soon became apparent that the entire approach was a failure, largely because:

- Too many Unix files were required for each property list group, resulting in both run-time and storage costs.
- Representing all property values as strings was too costly in both run-time and storage; a type-structure was obviously necessary.
- Property name strings were stored twice, resulting in significant wasted space.

7.2.1.2. Second failure

The next implementation concentrated on eliminating these flaws. Property values became variant records; the tag field indicated the data type, and the data field could then be interpreted directly as an integer, or indirectly as a string (via a mapping database). The use of a record allowed us to include a protection field for each property, another important modification to the design.

The string-mapping databases bore the brunt of the changes. The string-to-identifier map became a flattened tree structure, designed to perform efficiently when small, in recognition of the small number of entries expected. Further, the strings were stored using the "chunked" string storage package, eliminating the need for storing multiple copies of some strings. The string-mapping sub-databases were modified to use the same block-caching scheme that the B-Tree package used, and the block-caching scheme was modified to "multiplex" several apparently distinct sets of cached blocks onto one Unix file. In this way, a property list group could be stored entirely in one Unix file instead of four, which reduced the file-opening overhead by 75%. A substantial amount of run-time was saved by the use of lookaside caches for both string-mapping sub-databases.

We had solved the three serious problems discovered with the first implementation. However, performance was still inadequate:

- There was still too much disk I/O; we were using data structures, such as B-Trees, intended to minimize disk I/O in large databases, instead of realizing that our databases were fundamentally small and should be dealt with entirely in memory.
- We lacked an integrated string mapping mechanism; we could map from strings to identifiers, and from addresses to strings, but not between identifiers and addresses. We could not use the information in one of the string-mapping caches to satisfy lookups against the other one.
- The need for inverted indices had become apparent, but we could not easily integrate them with the basic database.
- It was clear that almost all property names, and many string property values, would be used in more than one property-list group. It made no sense to store thousands of copies of strings such as "owner" or "Fred" if we could store just one.
- It became clear that an even richer type structure, including real, boolean, time, and protection values in addition to integers and strings, would be an appropriate addition to our design.

Only a completely new approach would resolve these issues, and we proceeded to the implementation that will be described in section 7.3. We will first try to summarize some of the principles that can be extracted from our experiences, both successes and failures.

7.2.2. Achieving performance

To build a high-performance implementation of a property database system, one should follow these principles: understand and exploit locality of reference; use caches, hints, and pre-computed answers; cluster disk I/O; share string storage; and defer updates. We will expand upon these principles.

We are primarily concerned with reducing elapsed time, so we are usually willing to spend processor time to avoid disk I/O. We do not neglect CPU costs, but reducing them is only useful if it improves the response time of the system.

Understand and exploit locality of reference: This is the "meta-principle" behind the other principles, which are more specific instances of exploiting locality. One must understand, through intuition, analysis, and measurement, what the locales of reference are. Efficient implementations of file and directory systems depend on accurate characterizations of client file reference locality. For example, one such study demonstrates that there is a high locality of reference to files [Maju84].

In a directory system, the individual directory nodes are clearly strong locales of reference. Most systems exploit this through the concept of a "current working directory" in the user interface, since it relieves users of having to type long path names. It can also short-circuit the interpretation of path names. It is perhaps less obvious that this particular locality of reference implies that caches should be associated with individual directories, not the entire directory system.

The "current working directory" concept applies to one directory at a time, but a user will probably be using several directories more or less simultaneously. For example, a Unix user debugging a program might be executing commands out of three different directories, loading text editor macros from another, C header files and object libraries each from two more, and program sources from the current working directory. Many of these files will be accessed using absolute path names, so a half dozen more directories will be repeatedly searched to interpret the paths. Therefore, this user will have not one, but perhaps 15 locales of reference at once. A high performance system must exploit this "dispersed locality"[2].

Directories are almost always small. In typical multi-user systems with hierarchical directory structures, the average directory has about 20 entries, the median directory fewer than ten (see section 8.2). This means that it is a mistake to represent individual directories in data structures, such as B-Trees, designed for external storage of large databases. B-Trees minimize access costs to *single* records in a large database; a directory is a small database with a strong locality of reference, and we must minimize the cost of accessing the entire contents of a directory.

We have no experience with multi-version property-based file systems, so there is no observational evidence that property-list groups in such systems also tend to be small. However, our intuition is that most "multi-version" files will have just one version, and few will have more than several dozen.

Use caches, hints, and precomputed answers: To make a database implementation run fast, don't use it. Appropriate caches allow us to avoid most disk reads. Investing significant processor time to maintain a cache may well avoid comparable processor costs in I/O overhead and context-switching, in addition to an order of magnitude savings in elapsed time.

---

[2]This idea is slowly creeping into Berkeley Unix, first with the use of hash tables in the *csh* command interpreter to short-circuit command lookup, and recently with the use of various caches in the kernel file system.

The trick is to know what to cache, how much to cache, when to remove something to cache, and how to keep the caches honest. These are mostly static choices that depend on understanding what is stored and how it is accessed. Not surprisingly, the guiding principle is to anticipate locality of reference. For example, by treating a directory as a single, small, database, and reading the entire database into memory whenever we want to use any part of it, we are effectively prefetching a large cache block. Locality implies that such a cache will have a high hit ratio.

Cluster disk I/O: When we cannot avoid the distasteful necessity of using external storage, we should at least get it over with quickly. This does not so much mean minimizing the amount of data transferred as it means minimizing the number of disk operations.

There is a tremendous advantage to writing a given amount of data in one operation, as opposed to a series of short buffers. Suppose we have a data structure with eight-byte records; if we intend to update more than about 2% of the records in a file, it is faster to read and re-write the entire file. A similar ratio applies for read-only transactions. Moreover, this ratio is for sequential, i.e. highly localized, I/O; if access to individual records is randomly scattered, the break-even ratio should be lower[3]. The cost of prefetching an entire directory is no more than, and probably less than, the cost of fetching just those records necessary to satisfy the simplest query.

Share string storage: String values are repeated because property names are shared by many files (it is seldom useful to have only one instance of a property), and because string-valued properties often draw their values from a small pool. For example, there are fewer "owners" than there are files. One problem with our first implementation was that we often stored many copies of a string value. This is costly for two reasons:

1. It increases the amount of externally stored data. This has the direct cost of using up disk space, and it indirectly increases run-time costs, by requiring more data to be transferred.

2. Since there is nothing to connect the multiple copies of a string (if there were, we wouldn't need to store them all), each copy results in a separate cache entry.

We should try to store only one copy of these two kinds of string values.

The same principle does not apply to directory entry names. First, entry names are rarely repeated; in a typical Unix system, out of 66061 names, 35749 (54%) were unique and none was repeated more than 299 times. Second, whenever we operate on a directory, we *always* reference at least one of the entry names. We don't want to have to make additional disk accesses just to get the entry names, and we can cluster them together at negligible cost in storage space.

Defer updates: If a per-directory database is updated, it is likely that the same database will shortly be updated again; we often change at once many or all of the properties of an entry or version. These grouped updates would be costly if performed step-by-step; we can improve response time by deferring disk I/O until we believe that the group is complete.

Deferring updates presents some complications. We would need a failure-atomic implementation to guarantee consistency if the server crashes after acknowledging the update but before committing it to disk, and since the database may be shared by several clients, the locking mechanism must interact with the deferred update policy. We also need a way to decide when to commit the update when no lock is held. Although ours is not a transaction-based system, in

---

[3]These ratios were discovered using a simple test program on a 4.2BSD Vax/Unix system; they are meant only to be indicative.

such a system one must obviously commit a set of deferred updates upon committal of a transaction that includes them. In a non-transaction system, we have more latitude, but also less information: we can commit the updates after a certain number have been deferred, when the database is ''closed,'' or when no activity has taken place for a while.

7.3. A high-performance implementation

After the disappointing performance of the trial implementations described in section 7.2.1, we started on an implementation based on the performance principles of section 7.2.2. The result is an implementation that exhibits respectable performance. Access times are less than thrice those measured for Unix, and storage utilization is small compared to internal fragmentation losses.

Because this implementation runs as a client of Unix, rather than on the bare hardware, it does not fully realize its potential performance. Stonebraker discusses some of the problems with operating system support for database functions [Ston81]. However, it would have been much harder to develop this implementation without the scaffolding provided by Unix.

We refer to this implementation as PLDIR, for "Property List DIRectory system." PLDIR supports the complete design of chapter 6 except that, of the four string data types, only the two "atom" types are supported. The additional mechanism needed to implement subrange queries on "string" data types could be added quite easily, but perhaps not with sparkling performance.

7.3.1. Architectural overview

PLDIR is composed of three radically different kinds of database: a set of B-Trees for inverted indices, a central database to store string data, and a collection of databases that each store the information for one property-list group (i.e., for one directory node). Each of these database technologies was chosen to support a particular pattern of access and a particular database scheme.

The inverted indices are orthogonal to the rest of the system, so we will leave their description until later. The per-node and string databases are intimately related, and it is important to understand how.

Strings, for both property values and property names, are stored in a central database. The purpose of this database is to provide a fast, invertible mapping between strings and compact identifiers, called string-IDs. It is similar to the LISP concept of "interning" an atom name, and so it is called the INTERN database. This database is centralized to avoid the storage costs associated with keeping multiple copies of string values.

Each property-list group is stored in its own database (a single file of the underlying file system). In these, we store all the property records for the directory, and the names of the entries.

The INTERN and per-node databases are connected through the property records. Each record contains two key fields, which are compact identifiers for the property name (its string-ID) and entry name. It also contains a value field; if the value is of a string type, then this field is interpreted as a string-ID and the actual value is in the INTERN database.

7.3.2. Property-list group database

We now describe the per-node databases in more detail.

We chose to use a separate database for each node for several reasons:

- We exploit the logical locality of reference by achieving physical locality of reference.
- PLDIR can read or write the entire directory in one operation.
- It is easier to conceptualize such things as caches and locks if there is a single underlying entity associated with each directory.
- Storage allocation is easier, since the low-level file system provides dynamically-expandable "segments.*'
- If a node's data node were stored as part of a larger database file, then it would be difficult to write a portion of the database that has expanded.
- The node database files are usually quite compact, as shown in section 8.3.2.
- Two clients accessing separate directories do not interfere with each other[4]. A separate process can be used to serialize shared access to each active directory.

By storing the entry names in each directory, rather than as identified strings in the INTERN database, we minimize I/O, because the most frequent operations on directories are reading, writing, and listing the entry names.

Use of a separate database for each node also improves the overall reliability of the system, by providing a * 'firewall'' between the nodes across which failures cannot propagate. A single failed operation, or processor crash, can at most damage the directories that are actively in use. If all the directories were collected into one database, a single failure might corrupt the entire system. This firewall allows debugging without involving any innocent bystanders.

7.3.2.1. Structure of the per-node database

We use different representations of the per-node database for external storage and for internal manipulation. The database contains several sub-segments that may expand in size, so internally we allocate separate memory regions for each segments. The external representation must be compact, but internally we can allocate larger segments than necessary, accommodating growth without continual storage reallocation. The internal representation also uses auxiliary data structures to improve response time.

The external representation starts with a header describing the layout of the two other segments. Next is an array of fixed-size property records, followed by a sequence of entry name records; since these contain strings, they are variable-length. While a directory is open, a sequence of update log entries may be appended to the file; if a crash occurs before the directory is re-written, the log allows reconstruction. Reconstruction can be deferred until the next time the directory is opened, so the system can be available relatively soon after a crash recovery.

The internal representation, in addition to a slightly reformatted image of the external representation, includes two auxiliary tables to speed access to the entry name records: a hash

[4] They still share access to the INTERN database, but as we shall see, this is less important because updates to this database are infrequent and irreversible.

table for rapid mapping from entry names to compact entry identifiers, and an index array for the inverse mapping. It would be quite expensive to construct the hash table each time the database is read, because to hash each name involves scanning each byte of the string. Therefore, the hash function allows partial precomputation of the hash value, and this "pre-hash" value is stored in the external entry name records. The final stage of the hash computation, reduction of the pre-hash value modulo the size of the hash table, can be done quickly, once PLDIR has chosen the size of the internal hash table. This saves time when reading the database, and also makes it possible to rapidly rebuild the hash table if it becomes too full.

73.2.2. Algorithms

The algorithms used in accessing the per-node database are simple, since it is small and stored internally. Most of the data structures allow effectively constant-time access through hashing or indexing, or are small enough for linear search.

The property record array is sorted, first on entry identifier and then on property name identifier. This groups all the properties of an entry together, so that the linear search used to iterate over the properties of a single entry has optimal running time, once started at the first entry. We use binary search to find specific property records; careful coding of the binary search is central to performance.

Queries over the entire directory node are done by repeatedly invoking the binary search mechanism, once for each property list. While an additional auxiliary data structure, such as a copy of the property record array sorted on property name identifier as the primary key, would allow a faster algorithm, the frequency of such queries probably does not justify the added cost of constructing such a table.

None of these searches require either string comparisons, or repeated accesses to the INTERN database. Once the *'target" key has been constructed by mapping a single property name onto a string identifier, we are simply performing equality comparisons on integers.

The use of inverted indices complicates entry deletion, because we number the entries sequentially and use these numbers for part of the index key. We do not want to update all the index entries for the entire directory when an entry is deleted; the solution is to leave dummy entries in the entry name table, to be recycled when new entries are added[5].

7.3.2.3. Caching **and hints**

There is little need for caching in the module that manipulates the per-node database, since everything is in memory. We use a simple "hint" to significantly reduce the number of binary searches executed, based on a "locality of reference" observation: the likely reference patterns include repeated references to a single property, and references to a contiguous sequence of properties, the latter because all the properties on a property list are contiguous in the sorted array. When we search for a property record it is likely to be either the most recently used record, or the successor of that record, so we remember the array index of the last successful search, and before executing a subsequent binary search, we check both the indicated record and its successor. Although this check adds a few percent to the cost of an unprecedented search, it saves almost all of the search cost when it works, and on the balance comes out ahead.

We also make use of a hint to speed the iterator mechanisms. The iterator key value (opaque

[5] **We did not actually finish implementing this.**

to the client) passed between client and server carries not only an accurate representation of the current state of the iteration, but a possibly inaccurate hint that allows the server to bypass the binary search, if the database has not been updated in the interim.

7.3.3. String database

The INTERN, or string storage database, must:

- Rapidly map strings to compact identifiers
- Rapidly map identifiers to strings
- Conserve storage by avoiding multiple copies of string values
- Support both case-sensitive and case-insensitive strings
- Be highly reliable

There are only three operations on the INTERN database: add a string, lookup by name, and lookup by identifier. The add operation assigns and returns a unique identifier. The database associates a set of possible "types" with each string; for example, the string "Owner", although stored but once, can be looked up as either a property name or a property value. All three operations take a type, or set of types, as a parameter.

The add and lookup-by-name operations distinguish between case-sensitive and case-insensitive values. The INTERN database is thus logically two parallel databases; since case-sensitive strings are used infrequently, we elected to implement only one physical database, to amortize the overhead.

7.3.3.1. Deletion

We did not implement a delete operation. As a result, all operations are idempotent; if the add operation is repeated for a given string, it returns the same identifier and does not modify the database. Idempotency is helpful in providing a reliable service without much overhead; since the string database is centralized, this is important.

Once an entry is in the database, it stays there. This makes caching entries extremely easy, since there is no cache invalidation problem. Caching is critical to maintaining reasonable performance for the INTERN database, since otherwise there would be a great deal of random access to the disk image of the database.

The primary reason for the lack of a delete operation is that it would have taken a lot of work to implement it efficiently; it is not conceptually difficult. We elected not to reference-count the entries, since the repeated write-accesses to the INTERN database would eliminate the advantages of caching[6], so to determine which entries are deletable we would use a garbage-collection mechanism. This is simple in principle, since string identifiers cannot be held outside PLDIR, and so all the active references can be located. However, garbage collection complicates the implementation by requiring a way of searching the caches for references that are not elsewhere active. If strings are garbage-collected, the "address space" used by inactive string identifiers should be reclaimed. This can be done by keeping a "free list" of such identifiers.

---

[6]Reference counting also tremendously complicates reliability. A reference count must be right; if it is low, then active data is lost, while if it is high, then inactive data is not reclaimed, and one is forced to use garbage collection.

7.3.3.2. Database structure

INTERN has three sub-databases: "MainFile" for string storage, "IndexFile" to map string identifiers onto addresses in string storage, and "HashFile" to map, via a hash table, strings onto storage addresses. Each sub-database is a separate Unix file. This provides three "address space" segments; since address space expansion is provided by the Unix file system, the implementation is quite simple.

Records in the MainFile are variable-length, and contain a character string, the associated identifier value, and a "type mask" recording the set of types for which an add operation has been done for this string. Because the identifier value is included in the MainFile record, the IndexFile and HashFile can be reconstructed as long as the MainFile is intact.

The IndexFile is simply an array, indexed by string identifier, giving offsets into the MainFile for the corresponding string.

The HashFile starts with a header record, describing the size of the hash table and the number of slots in use. Following the header record is an array of hash table entries. A hash table entry, in addition to storing an offset into the MainFile for the corresponding string, contains a "PreHash" field, which will be described shortly.

INTERN maintains a sophisticated cache of recently used entries. This cache significantly reduces the likelihood that a reference to external storage is required, and so dramatically improves performance that it is worth spending some CPU time managing a large cache.

7.3.3.3. Basic Algorithms

The following are sketches of the algorithms used for the operations on the INTERN database. These descriptions ignore use of the cache; the cache algorithms are described in the next section.

Lookup by string: A hash function is applied to the string to obtain a "prehash" value. Different hash functions are used for case-sensitive and case-insensitive strings. The prehash value is reduced modulo the size of the hash table to get an initial slot number. We use linear probing to resolve hash collisions. Since the cost of each probe is significant[7], we use the "PreHash" field in the HashFile records, mentioned earlier, to avoid most potential MainFile reads. The string referred to by the hash entry being probed can only match the search string if the prehash values match; only in rare cases will distinct strings have the same prehash value.

Once the appropriate MainFile record is found, the type masks are compared, and if they are compatible, the identifier found in the MainFile record is returned.

Lookup by identifier: The identifier is used as an index into the IndexFile. The offset found in the IndexFile is used to retrieve the appropriate record from the MainFile, the type masks are compared, and if they are compatible, the identifier found in the MainFile record is returned.

Adding an entry: We first search the database to see if the entry already exists; if it does, we update the type mask in the MainFile record, if necessary, and return the identifier found in that record.

---

[7]Each probe requires a system call to read the next record from the hash file. Since we use linear probing, this does not require a disk read to the HashFile per probe, because many consecutive probes map onto the same disk block, and Unix buffers disk blocks in the kernel. However, when looking up a string (but not when finding a slot to add a new entry), each probe requires reading a record from the MainFile, and these reads are not sequential.

Otherwise, we assign a new identifier (derived from a counter in the HashFile header) append a new record to the MainFile, and append a record consisting of the MainFile offset to the IndexFile.

To update the HashFile, we compute an initial hash slot number, and then use linear probing until an empty slot is found. A new HashFile record is created and inserted in that slot.

Resizing the hash table: To reduce hash collisions, we require that at least half of the hash table slots be empty. When an identifier is allocated for a new entry, we may find that there are not enough slots left We rebuild the hash table by creating a new table with twice as many slots, and reinserting all the records.

We could read all the records in the MainFile and insert each one into the new table, but this is costly, since it requires recomputing the hash value for each string. Instead, we use the precomputed PreHash values, stored in the existing HashFile records, which need only be reduced modulo the size of the new table. To minimize disk I/O, we read the entire old table into memory, and build an image of the new table in memory before writing it to the disk. This works if there is enough real memory; access to the hash tables is by design non-local, and if the hash table is too large, a more sophisticated approach would be necessary to avoid memory thrashing.

7.3.3A Cache Algorithms

INTERN relies on a cache to improve response time. Since database entries are never deleted, it is reasonable to maintain multiple copies of this cache, reducing the tendency for INTERN to be a concurrency bottleneck.

The cache is managed using a least-recently-used replacement algorithm. It must be large enough so that, for example, listing all the properties of an object does not "sweep" the entire cache, but it must not be so large that searching the cache becomes too expensive. Perhaps a simple adaptive mechanism, which keeps track of the average cache miss ratio and disk response time, could keep the cache size near optimum; the current system uses a fixed size.

The cache can be searched using either a string or an identifier as a key. When the key is a string, we must avoid performing too many string comparisons. Therefore, a cache entry includes the PreHash value for the string. Before searching the cache for a string value, we calculate the PreHash value for the string. The search can then be based on a simple comparison of the PreHash value, instead of an expensive string comparison, until we find a likely candidate. Since we store PreHash values in the cache, not hash table slot numbers, the on-disk cache database can be restructured without affecting the validity of cache entries.

7.3.4. Inverted indices

The PLDIR system uses inverted indices to answer global queries. Such an index must efficiently support both updates of single entries and retrievals of all entries within a range of values. We chose to use B-trees [Baye72], since they provide initial access in logarithmic time, yet are sorted and so provide optimal access to subsequent values in a range. B-trees also map well onto external storage, since they have a large branching factor and take advantage of the block structure typical of magnetic disks.

There is nothing particularly novel about our implementation of B-trees. Since the algorithms, while relatively simple to describe, require lengthy and fairly intricate code, it took several months to implement and debug the B-tree package alone. This time would have been

better spent otherwise, but when we started we could not find a B-tree package for Unix that used external storage.

A production-quality system should include a more polished B-tree package. Our naive implementation of B-trees yields nodes that are almost half empty, which wastes disk space and increases the number of disk accesses. A more sophisticated algorithm for splitting full nodes could in many cases reduce the waste [Come79]. Also, our implementation is not especially crash-proof.

7.3.4.1. Database structure

Each time a property value is created or modified, PLDIR checks to see if an index exists for the property. If one does, a B-tree key is constructed and the B-tree is updated. Each B-tree is stored in one Unix file; we use the Unix directory system to map property names onto B-tree databases.

These B-trees contain only keys. Each key describes one property record and has, in decreasing order of significance, four components:

1. The data type of the property.
2. The property value.
3. The property-list group identifier.
4. The property-list identifier.

All of these components are represented as integers, using INTERN to map UIDs and property-list names onto compact identifiers. This means that the B-tree keys are a constant size, 16 bytes long, so the B-tree package must support these rather large keys.

When we are searching for a set of properties with a specific value or range of values, the B-tree package must allow us to retrieve a set of keys bounded by values that do not actually occur in the database. For example, to find the entries with a type of INTEGER and values between 10 and 20, we ask for all keys in the range between (INTEGER, 10, MINUID, MINENTRYID) and (INTEGER, 20, MAXUID, MAXENTRYID), where MINUID, MAXUID, MINENTRYID, and MAXENTRYID are appropriate constants. (When we search for values of an unordered type such as ATOM or PROTECTION, the second components of the two B-tree keys must be identical.) To find the keys lying outside a specified range, we search the B-tree for keys in the two ranges that form the complement of the specified range.

7.3.4.2. Caching

Although a B-tree requires only a few disk reads for access to a key, we can reduce the number of reads even further by using a cache to take advantage of locality of reference. Some keys can be found with no disk accesses at all, and many others with only one.

There are two kinds of cache used in the B-tree implementation: a large LRU cache of disk blocks, and several "hints" used to bypass repeated searches. One hint records the last leaf node visited; in the (frequent) case that successive accesses are made to the same key, or adjacent keys, it means that only one node need be searched if the hint is valid, not O(log(*number of keys*)) nodes. This saves considerable CPU time[8], and it also avoids sweeping the LRU cache with the internal B-tree nodes.

---

[8]Although binary search is used to locate the appropriate child of each internal node, the 16-byte keys require a fairly expensive comparison function.

The LRU cache is a write-through cache of B-tree nodes, and is logically a layer between the B-tree package and the underlying Unix file system. We cannot rely solely on the Unix kernel disk cache, because:

- The replacement algorithm is global to all files in use on the Unix system, so the blocks we consider "active" might not be retained in the kernel cache. We need only cache a small set of blocks (e.g., the B-tree root node and the most recently visited leaf node) to achieve reasonable performance, but these blocks must not kicked out.
- The cost of crossing between user mode and kernel to get at the cache, and of copying the data blocks in and out of kernel space, is excessive. We can get at a cache kept in our address space within a few microseconds, and need only cross the domain boundary for write-through.

There are some inefficiencies in duplicating the Unix cache; none are particularly significant.

Because of these caches, repetitive updates to the B-tree are almost entirely CPU-bound. Most time is spent searching a B-tree node for a given key. Originally, we used binary search, but because the keys used by PLDIR are so large (16 bytes), it is actually faster to use a clever linear search algorithm that need not look at the entire key. We also use another hint: in each B-tree node we record the key index of the last successful search of that node, so that on both repetitive updates and subrange queries a full search is seldom required.

Execution profiles of the B-tree code indicate that careful tuning might improve performance by an additional 10% or so, by replacing several heavily used function calls or code sequences with single VAX machine instructions or in-line hand-coded assembly language. This would make the resulting code less portable and harder to maintain.

7.3.4.3. Reliability

If the keys stored in the index B-trees were the only accurate copies of the information they encode, then the B-tree implementation would have to be highly reliable and crash-proof. In PLDIR this information always exists in the per-node databases (and we take some effort to keep these accurate), so we can always reconstruct an index if it is damaged.

We must still be able to determine if a B-tree is damaged. Assuming that we can detect if a disk page has been corrupted or incompletely written, we can guarantee that no keys are ever lost if we do not at any time keep the only copy of a valid key in volatile memory[9]. Since leaf nodes are distinguishable from internal nodes, we can rebuild the B-tree by scanning the leaf nodes, extracting the keys, eliminating duplicates, and building a new B-tree. If any of the leaf nodes are detectably corrupt, then we must instead rebuild the B-tree from the per-directory databases.

This method does not by itself prevent a key being deleted at the time of the crash from showing up in the reconstructed database. We could use an intentions list to do this, and to avoid losing pending updates.

The current implementation includes none of this. This is partly because the additional work to provide these facilities is beyond the proper scope of the thesis, but also because Unix does not really allow us to efficiently guarantee the order of disk page writes.

---

[9]We must not overwrite a disk page *P0* containing a set of keys *K* with a page *P0'* such that the set *K* is not fully present in the union of the keys in page *P0'* and some distinct page *P1*.

7.3.4.4. Concurrency

A simple B-tree implementation has an inherent concurrency bottleneck: all update operations must lock the root node of the B-tree, since any change to a leaf node can potentially require modifying the root. There is the additional problem that the caches upon which the efficiency of this implementation relies can be invalidated by any update. Either we need a cache invalidation mechanism, or we must serialize access at a level above the caches.

We speculate that the best approach is to use a single server process for a B-tree (one process could manage more than one B-tree). Communication between PLDIR and this server would then become the concurrency bottleneck. However, the server can defer B-tree updates, once it has written them to its intentions list, allowing other PLDIR activity to take place in parallel with B-tree operations. If the server checks the intentions list for pending updates whenever a query is performed, and as long as the intentions list does not get too long, it need not manipulate the B-tree immediately.

7.3.5. Atomic Transactions

A simple analysis convinces us that PLDIR can be made to support atomic transactions.

The INTERN database trivially supports atomic transactions: even if a transaction aborts after entering a new string, we do not care if the string remains in the database. We would have to synchronize garbage collection with transactions, if we implement a deletion operation.

The inverted index database can be extended to support transactions by tagging pending updates in an intentions list with their transaction identifiers (TIDs). Until a transaction commits, its pending updates could not be written to the B-tree, nor could they be returned in answer to a query. To commit a transaction, we would simply mark the associated "intentions" as committed. The committed intentions could then be transferred to the B-tree as time permits.

The per-directory database implementation requires that concurrent access to each database be mediated by a single server process. Once an update transaction starts, the server process must lock out further update transactions until the first one terminates. The server would apply the updates to a complete shadow copy of the database and associated caches. When the transaction commits, the real database is replaced with this shadow database, atomically written to stable storage. A more sophisticated implementation could lock and shadow specific property lists instead of the entire directory.

Since a single transaction can potentially involve the INTERN database, many inverted indices, and several per-directory databases, the individual database commits must be coordinated via a global intentions list. The intentions list mechanism depends on highly reliable nonvolatile storage that can be updated quickly. A variety of atomic transaction systems use this or a similar mechanism [Lisk83, Lamp81, Spec83].

7.3.6. Implementation statistics

A facility such as the PLDIR system inevitably makes a file system implementation more complex. It is fair to ask if the additional complexity is an excessive burden, both in terms of the amount of source code, and the additional run-time memory requirements (CPU time and elapsed time requirements will be discussed in section 8.3.2).

These statistics cannot be exact, because it is hard to account for the effects of:

- Code for debugging and testing, including "dead" code
- Code shared between PLDIR and other implementations
- Code required to bridge the gap between the C language and the RPC facility we used
- Incompletely implemented features
- Run-time variability in dynamic data structures

7.3.6.1. Source code statistics

Source code sizes are given for heavily commented code, with lots of white space. For rough comparison, a representative module of PLDIR code is about 55% non-comment source code if one counts lines, or 65% non-comment source code if one counts characters. For a representative Unix kernel module, the corresponding proportions are 80% and 84%, respectively.

For each major subsystem of PLDIR, we give the number of source modules, the total number of source lines, and the size of the source in Kbytes:

Subsystem	Modules	Lines	Kbytes
B-tree:			
Disk page cache	7	919	19
B-tree proper	9	2440	63
Total	16	3359	83
INTERN database	8	1716	41
Per-directory database	13	3450	81
Miscellaneous	30	3095	65
Total for PLDIR	**68**	**11620**	**270**

Included under "Miscellaneous" are routines for interfacing to the RPC system, some data abstractions, simplifying memory allocation, debugging, and formatting data to be viewed by humans.

Of the source code described above, almost all can be used for both the directory system and a file system. A few of the modules used to implement the per-directory databases would have to be modified slightly, to reflect the different interface, as well as some of the "Miscellaneous" modules. A file system would also include additional code to manage the storage of file contents.

7.3.6.2. Object code statistics

The entire PLDIR system, when compiled with the standard peephole optimizer, comes to about 47K bytes of VAX machine code. In addition, it includes about 12K bytes of initialized data; at least half of this is devoted to the error message strings returned upon exceptions. At run time, a substantial amount of memory is allocated dynamically; see section 8.3.2 for details.

7.3.6.3. Comparison to Unix

As a crude comparison, it is interesting to see how much code Unix requires to perform the functions roughly analogous to what PLDIR implements; essentially, this means directory and "inode" management. It is not possible to identify Unix kernel modules as entirely relevant, or

entirely irrelevant, to these functions, but in the 4.2BSD implementation of Unix three code modules are clearly related. Together, these three files account for 2276 lines of source code, or 57K bytes. They compile to about 9K bytes of VAX object code; it was not practical to determine how much run time data storage is allocated to these modules, but it is presumably on the order of 10K bytes. Overall, a typical 4.2BSD (VAX) kernel contains over 200K bytes of object code. Thus, the PLDIR implementation requires about five times as much code as does the analogous part of the Unix, but would not significantly increase the total size of the resident system.

7.3.7. Possible improvements

There are a few potentially beneficial modifications to the PLDIR implementation that we have not yet tried. None are likely to produce dramatic performance improvements, but they might provide noticeable results.

We build a contiguous internal image of the per-node databases before writing them back to disk, so that only one kernel call is necessary to perform the write. 4.2BSD Unix provides "scatter/gather" read/write system calls; it is possible that by using the "gather write," we can maintain the low kernel overhead of the single system call, and avoid the additional cost of allocating memory and copying data to build the contiguous image.

There is room for tuning the various caches in the system. A larger cache size might improve the hit ratio, at the expense of costlier lookup and replacement processing. A moderate increase in the size of some caches might actually be a mistake, since processing costs would increase, yet "cache-sweeping" might keep the hit ratio from rising.

One possible improvement would be to separate the INTERN cache into two sections: one for property names, and one for string values. Alternatively, the cache could be broken into a small set of "active" entries and a larger set of "inactive entries"; the first could be managed LRU and searched quickly, while the second could be managed FIFO and updated quickly (i.e., as a doubly-linked list).

7.4. A front-end implementation for Unix

We asserted in section 2.5 that we can derive substantial benefit from uniformity in file system interfaces, even if some of the implementations are incomplete. Uniformity allows us to write portable programs and programs for use in heterogeneous distributed systems, if we restrict ourselves to a small class of operations. While a complete solution to the heterogeneity problem requires extensible property support, as argued in chapter 10, a uniform interface allows us to make real progress.

We take as a particular example the Unix file system. The Unix file system is deficient in several important ways:

- The set of properties is small and closed.
- There is no inverted index support for queries.
- Only a few of the supported properties can be directly modified.

Except for the last one, which could be declared to be a policy decision against allowing explicit modification of certain properties, these deficiences cannot be papered over simply by changing the interface syntax.

There are some failings of the Unix file system that *can* be hidden under a thin "cosmetic" layer:

- The interface providing access to properties is peculiar to Unix.
- A different system call is required to update each of the few properties that can be directly modified.
- The values of certain properties are densely encoded.
- Property types are not explicit.

To demonstrate that these failings can be overcome, we implemented a subset of the interface defined in chapter 6 as a layer above the Unix file system. We will refer to this implementation as UFE, for "Unix Front End." Without significantly compromising the performance of Unix, UFE allows a single program can be used with either PLDIR or Unix. For example, a directory-listing program (like the Unix *ls* command) written for use with PLDIR could be used with UFE as well. To sophisticated applications the difference in implementations would not be transparent, but to a large class of programs, it would be.

UFE does not use any non-volatile storage of its own, nor does it require any modification to Unix. It is simply a package of functions that can be linked with any program to run locally under Unix, or with a server process for use in a distributed system.

UFE is an implementation of a directory system, not of a file system. The line drawn in chapter 4 between directory system and file system is blurred in Unix, and UFE does nothing to clear up the distinction.

7.4.1. Implementation

7.4.1.1. Property Access

Unix stores these file properties:

- Access, Creation, and Modification times
- (Device, Inode) pair uniquely identifying the file
- Identifiers for the owning user and access group
- The number of links to the file
- The size of the file, in bytes
- A device identifier for "special" files
- A set of "mode" bits, including protection and type information

We elected to treat the tail of a path name for the file as a property. Also, we treat the "type" bits of the "mode" as a separate property with a string value, indicating if the "file" is a directory, a regular file, or some other Unix object.

All of these properties (except the name) are read as a group using the ***stat***() system call. The ***GetProp***() function is implemented in UFE by a fixed mapping between the known property names and corresponding fields in the structure returned by ***stat***().

Since ***stat***() is a fairly expensive operation (it requires a directory lookup in addition to the potential external storage access), and because a typical application is interested in more than one property of a file, we keep a small (eight entry) LRU cache of `stat` values. Unix could modify the underlying values while we keep old copies in this cache, so we arbitrarily invalidate any cache records more than 2 seconds old. This is a compromise between efficiency and correctness; some applications will have to use locking mechanisms to avoid inconsistencies.

Implementing ***PutProp***() is another matter. Unix provides direct update access only to a small set of properties:

- Access and Modification times
- User and Group IDs

- Mode bits (but not the type bits)

It is fairly easy to modify the length of a file, but we do not allow this as a property operation. Thes five modifiable properties are, under Unix, updated using four different system calls. UFE maps ***PutProp***()s on these properties to the appropriate system calls, and returns an error if an attempt is made to modify any other properties.

Since Unix does not support protection of individual properties, UFE returns via ***GetProp***() property protection values that indicate whether a property is modifiable or not. ***PutProp*** ignores attempts to change property protections. Unix additionally restricts some modifications, such as changing the user ID, to a distinguished user; attempts to violate this restriction are reported as errors.

UFE implements iteration over the properties of a file in a straightforward manner. An iteration simply maps onto stepping an index through the table of known properties.

7.4.1.2. Directory Access

In addition to the property access operations, the directory system design includes operations on directories and directory entries. UFE maps most of these onto equivalent Unix operations, but a few operations cannot be mapped exactly. For example, it is impossible, under Unix, to simply create a directory entry. UFE must instead create an empty file, so that the entry has something to refer to.

Iterating over the entries in a directory (i.e., doing a directory listing) maps fairly well onto a set of library routines provided for that purpose with 4.2BSD Unix. To avoid a complicated implementation, we chose not to handle the situation where a server process crashes and recovers during an iteration.

7.4.2. Summary

UFE works. It is compatible with programs written for use with PLDIR, and its performance is insignificantly slower than Unix. The implementation is economical, with about 1800 lines of commented source code, compiling into about 23K bytes of VAX machine code.

7.5. Using a relational database

We do not believe that currently available relational database systems can provide the performance of PLDIR. The arrival of a suitably fast relational database system would not make our file system design obsolete; rather, our design would indicate the relation scheme to be used. An efficient relational database would make a perfectly good basis for implementing the design of chapter 6.

To demonstrate the practicality of such a system, we implemented our directory design using the INGRES relational database [Ston76]. We refer to this implementation as PLING, for "Property Lists based on INGres." We cannot draw any interesting performance results from this exercise, because:

1. INGRES is not very fast[10].

[10]We used the "free" INGRES that comes with Berkeley Unix; it is known to be slower than the INGRES sold as a product.

2. Communication between a program and INGRES is cumbersome and extremely inefficient.

3. INGRES is missing a few simple features that would allow a much more efficient implementation of our design.

We show that were these problems to be resolved, there are no obvious obstacles to using a relational database.

7.5.1. Architectural Overview

PLING presents the same client interface as PLDIR, but the underlying data is stored entirely in the INGRES relational database. PLING simply translates between PLDIR-style operations and operations on the relations in the database.

In PLDIR, the code implementing client operations has direct access to the underlying data structures via a series of procedural layers; everything is done in a single process. INGRES does not allow direct access to the database; client code such as PLING must run in a separate process, and communicate with an INGRES server process using IPC. This is done using a package called EQUEL, which consists of a library of communication routines, and a pre-processor that turns programs written in a hybrid language into C code.

Communication between an EQUEL program and an INGRES server process is carried out across Unix "pipes." This is the primary source of inefficiency in PLING, especially because a poor coding is used on this channel. We will expand on this problem in section 7.5.3.

7.5.2. Relation Scheme

The core of PLING is the relation scheme used to represent directories and properties. Six relations are used; they are listed in Figures 7-1, 7-2, and 7-3.

The first two relations, DIRECTORY and PROPS, store the structural information (see figure 7-1). They take the place of the per-node databases in PLDIR, but in this case the information is centralized. The DIRECTORY relation gathers entries together into directory nodes; the PROPS relation contains all the property values.

The next three relations (see figure 7-2) are used to convert between character strings and compact internal identifiers (integers). ENTRIES, PROPNAMES, and STRINGS are used for directory entry names, property names, and string values, respectively. The latter two effectively replace the INTERN sub-database of PLDIR; in PLDIR, the per-node databases store the entry name maps that here are kept in ENTRIES.

The last relation, MAXIMA, is used to generate unique internal identifiers for use in representing names, string values, and other things (see 7-3). This is perhaps not the most direct way of generating unique identifiers, but it suffices.

An example will illustrate how these relations are used. Suppose that we want to find the "ModifiedTime" property for the file `/foo/bar`. We begin by asking INGRES to look in the ENTRIES relation for the tuple with **entryname** = `foo`, and we extract from that tuple the **entryid**, say 37. The directory ID for the path-name root is a well-known value, 1. We use these two identifiers as a key to retrieve a tuple from the DIRECTORY relation, and extract the **direntid** from this tuple, say 91.

We must now retrieve "EntryType" and "ObjectID" properties of the file `/foo` to deter-

DIRECTORY relation

- Contains a tuple for every directory entry, mapping from *(directory ID$_t$ entry ID) => entry-in-directoryID*
- Attributes:
  - dirid: Directory ID
  - entryid: Entry within directory
  - direntid: Global entry ID
- Key: (dirid, entryid)
- Stored as ISAM because multiple tuples will have the same primary key (directory ID).

PROPS relation

- Contains one tuple for each property
- Attributes:
  - direntid: Global entry-in-directory ID
  - pnid: Property Name ID
  - ptype: Property Type
  - prot: Property Protection
  - value: Property Value
- Key: (direntid, pnid)
- Stored as ISAM to allow range queries

Figure 7-1: Relations for INGRES-based implementation: DIRECTORY, PROPS

ENTRIES relation

- Global mapping between entry name and entry ID
- Attributes:
  - entry name: Name of a directory entry
  - entryid: Internal identifier for entry name
- Key: Either attribute can be used as key
- Relation and its inverted index are both stored as hashed files because only exact queries are done

PROPNAMES relation

- Global mapping between property name and internal property name ID
- Attributes:
  - propname: Property name
  - pnid: Property name ID
- Key: Either attribute can be used as key
- Relation and its inverted index are both stored as hashed files because only exact queries are done

STRINGS relation

- Global mapping between string value and internal string ID
- Attributes:
  - stringval: Character string value
  - stringid: Internal identifier for string value
- Key: Either attribute can be used as key
- Relation and its inverted index are both stored as hashed file because only exact queries are done

Figure 7-2: Relations for INGRES-based implementation: ENTRIES, PROPNAMES, STRINGS

MAXIMA relation

- Used to allocate new ID numbers for use in the relations above
- Attributes:
  - idname: Attribute name of ID
  - maximum: current highest ID in use for that attribute
- Key: idname

Figure 7-3: Relations for INGRES-based implementation: MAXIMA

mine if it is a directory, and if so, to obtain its directory ID. To retrieve the "EntryType", we query the PROPNAMES relation to find the pnid for that property name, say 3. We then use

the **direntid** and **pnid** (91 and 3) as a key to retrieve a tuple from the PROPS relation. We check the **prot** attribute of this tuple for access control, and the **ptype** attribute against an implementation constant to make sure that the property represents a character string. Finally, we use the **value** attribute as a key to the STRINGS relation, and retrieve the property value string.

We repeat this procedure to obtain the "ObjectID" property value, which can then be used to obtain a **direntid** for /foo/bar. Finally, we repeat the property-value procedure one last time to find the "ModifiedTime" property value we were seeking. Note that in this case, the property type is not a character string, so we directly interpret the **value** attribute instead of looking a string up in STRINGS.

To set a property value, we follow a similar procedure. When we add a new entry, entry name, property name, or string value, we must obtain a new internal identifier using the MAXIMA relations before updating the other relations. Otherwise, the connections between the six relations were all illustrated in the example.

7.5.3. Inadequacies in INGRES

The principal problem with INGRES/EQUEL is the use of pipes to communicate between the PLING process and the INGRES server process. This is inefficient because data must be moved between process address spaces, at least one context switch must be done each time a message is passed, and the data on the pipes must be parsed to locate message boundaries.

The way messages are coded for transmission on pipes is also wasteful. INGRES is designed to be used interactively, and its only command language has an English-like syntax. All messages must be converted into this syntax, and (much worse) must be parsed at the receiving end.

Another inefficiency is that INGRES string values are fixed-length. It is possible to have strings stored with trailing blanks removed, but when they are transmitted over a pipe all characters must be present. Although we would like to allow fairly long strings (for file and property names, and for property values), performance is noticeably better if the maximum length for strings is made fairly short.

A final deficiency in INGRES is that it can perform only case-sensitive string matches. It would not be difficult to modify the INGRES to allow case-insensitive matching, but this would not have proven anything. Therefore, PLING differs from PLDIR in its treatment of strings.

PLING performance is treated quantitatively in section **8.3.4.**

Chapter 8
Performance Analysis

Had we but world enough, and time
— Andrew Marvell

Response time means as much to 20th-century computer users as it did to 17th-century poets. The feasibility of a computer system depends on how much storage space it requires and how fast it runs. In this chapter, we present a performance analysis of the implementations described in chapter 7.

We begin by establishing a context for understanding the significance of property-operation performance. In section 8.1 we show that, in a typical Unix system, property operations are more frequent than file data transfer operations, and that they use more processor time. Performance of property operations is therefore central to file system performance.

In section 8.2, we present measured directory size distributions for a number of different systems. These measurements justify our assumption that directories are usually small. This both validates our design choices, and indicates what distributions should be used to measure performance.

Section 8.3 presents detailed performance measurements for the implementations of chapter 7. Finally, section 8.4 discusses the merits of possible alternative implementations.

8.1. Frequency of property operations in a real system

One of the lessons of the Reduced Instruction Set Computer (RISC) approach to processor design is that one should not optimize the performance of operations that are seldom invoked, if this is even slightly detrimental to the performance of common operations [Patt85]. Because it is possible to shift costs of database operations between read and update functions, a performance analysis of file property operations must take into account the relative frequency of property reads and writes, and the relative frequency of property operations and data transfer operations. If property reads dominate property writes, then it makes sense to tune an implementation to make reads run fast, even if writes run a little slower.

We collected data on the relative frequency of system calls on a 4.2BSD Unix timesharing system, a VAX-11/780, with about 1.5 gigabytes of disk storage, and a community of 120 active users, of whom typically a dozen are logged in at any given time during the day. The Unix kernel was compiled to profile the execution of each kernel function, providing both the number of times the function is called and the amount of processor time spent in the function or its callees. Profiling was done continuously for more than three days, so the results represent a typical mixture of daytime interactive use, and nighttime housekeeping. The profile was done using the *gprof* program [Grah82].

The resulting counts are shown in table 8-1. The first column categorizes the functions named in the second column; the categories are: data transfer operations; property read operations; property write operations; and miscellaneous operations that affect file properties. The next two columns show CPU spent in each function and its descendants, respectively; the sum of these is the total CPU time devoted to the function. The last column gives the total number of calls made to each function. A few miscellaneous operations are not shown, because they are essentially unused.

Category	Function name	Processor time spent in: self	descendents	Calls made
Data read (all)	*read*	149.36	4561.55	3291099
Data write (all)	*write*	68.92	1688.61	1218307
Data: total		218.28	4578.43	4509406
Data read (disk files only)	*read*			583030
Data write (disk files only)	*write*			130756
Disk file data: total				**713786**
Property read	*lstat*	24.43	5898.12	628637
Property read	*fstat*	12.48	23.14	133096
Property read	*stat*	3.39	910.75	97070
Property read: subtotal		40.30	6832.01	**858803**
Property write	*chmod*	0.45	37.59	4039
Property write	*chown*	0.04	11.48	1151
Property write	*utimes*	0.00	0.60	63
Property write: subtotal		0.49	49.67	**5253**
Property: total		40.79	6881.68	**864056**
Miscellaneous	*open*	16.71	3170.68	317742
Miscellaneous	*creat*	1.82	299.02	29966
Miscellaneous: total		18.53	3469.70	**347708**
File system: total				**1925550**
I/O: total		277.60	14929.81	5721170
All system calls: total	*syscall*	3724.30	29460.32	31832413

Table 8-1: Selected system call counts from a 4.2BSD Unix system

Most of the data transfer calls actually operate on non-file devices, interprocess communication, and network streams. The counts for reads and writes on disk files have been separated out; it was not practical to show CPU time separately for these operations. Only about 16% of the data transfer system calls operate on files. It was not possible to determine the proportion of property operations that operate on files, but it is reasonable to assume that almost all do.

Functions that write file properties are called much less often than those that read properties; 99% of the explicit property operations are reads. This is somewhat misleading, since properties are implicitly updated by Unix whenever a data transfer takes place, but properties are also implicitly read in many circumstances. It does indirectly demonstrate Unix's complete lack of user-managed property values.

The profile shows that property read operations are actually more frequent than data transfer operations. Moreover, although not clear from table 8-1, the ***lstat***() function accounts for more processor time use than any other system call; the three property read operations together account for 20% of the CPU time spent handling system calls[1]. Unfortunately, the profiling tools

[1]The developers of Berkeley Unix improved the performance of this operation in the subsequent version of their system [Leff84].

cannot show the elapsed time required to perform these functions, but they do show that the bulk of the time is spent doing file name translations rather than actually manipulating the properties. Note that *fsta*(), which operates on an open file identifier rather than a file name, is far less costly than *statQ* or *istatQ*.

The relative frequencies of various operations are summarized in table 8-2.

		As a percentage of	
Category	All calls	File system calls	File data transfer calls
File system	6.0%	100%	
File data transfer	2.2%	37.0%	100%
Property read	2.6%	44.6%	120%
Property write	0.02%	0.27%	0.7%
Total: property read or write	**2.7%**	44.8%	**121%**

The frequencies in this table are derived from table 8-1.

Table 8-2: Relative frequencies of system calls

8.1.1. Implications

These results imply that an implementation must emphasize the performance of property reading. They also indicate that property-based searching is central to the overall performance of a large file system; the surprisingly high frequency of property read operations implies that Unix suffers from its failure to support property-based search. On the profiled Unix system, the entire file system is searched at least twice a day, to support various administrative functions. Each of these scans accounts for over 100,000 *IstatQ* (property read) operations.

If such search functions are in fact useful, then a system that truly supports them will, overall, operate faster, even though some additional effort is required to maintain indices. Since property-write operations are relatively infrequent, they can absorb a large cost increase without disturbing the benefits of indices.

The relatively low frequency of explicit property writes in Unix does not necessarily apply to an extensible-property system. Nevertheless, our intuition is that reads will dominate writes.

8.2. Directory size distributions in real systems

We asserted in section 7.2.2 that a directory system should manipulate entire directories in memory, rather than piecemeal on disk. This is useful because directories are active locales of reference; it is possible because directories are "usually quite small." In this section, we show just how small they are, using data from a number of typical systems.

We drew our first samples from several Berkeley Unix systems with moderately large user communities. A simple program traversed the entire directory hierarchy and counted the number of entries in each directory. The program reports the number of directories found, the maximum depth of the directory hierarchy, and statistics on the number of entries, subdirectories, and symbolic links per directory[2].

[2] **At first it seems curious that the mean number of subdirectories per directory is nearly 1.0, but this is inevitable in Unix because each directory must have exactly one name: every directory, save the root, accounts for exactly one entry in some other directory (ignoring the "." and ".." entries).**

The results for five systems are summarized in table 8-3. Systems A and B are used primarily by single research projects, systems C and D are used by wider communities, and System E primarily stores "bulletin board" messages. Means and standard deviations are given for the number of entries per directory, but the large deviations indicate that the distributions are far from normal; see figure 8-1.

System	A	B	C	D	E
Directories	3035	4732	1866	3995	763
Files	66011	79923	40174	53493	49320
Maximum depth	11	10	10	10	8
Users	374	114	300	265	32
Entries per directory:					
Mean	21.75	16.89	21.53	13.39	64.64
Std. dev.	42.47	28.88	67.13	36.46	128.9
Median	9	8	5	5	11
Minimum	0	0	0	0	0
Maximum	683	424	1147	948	972
Subdirectories per directory:					
Mean	0.9997	0.9998	0.9995	0.9998	0.9987
Std. dev.	4.331	3.635	4.898	4.123	5.336
Median	0	0	0	0	0
Minimum	0	0	0	0	0
Maximum	119	107	108	112	120
Symbolic links per directory:					
Mean	0.1802	0.1942	0.0466	0.2308	0.0629
Std. dev.	5.374	3.022	0.5223	7.941	0.5620
Median	0	0	0	0	0
Minimum	0	0	0	0	0
Maximum	271	95	14	442	10
Dirs/User	8.11	41.5	6.22	15.1	23.8
Files/User	176	701	133	201	1541

Table 8-3: Distribution of directory sizes on several Unix systems

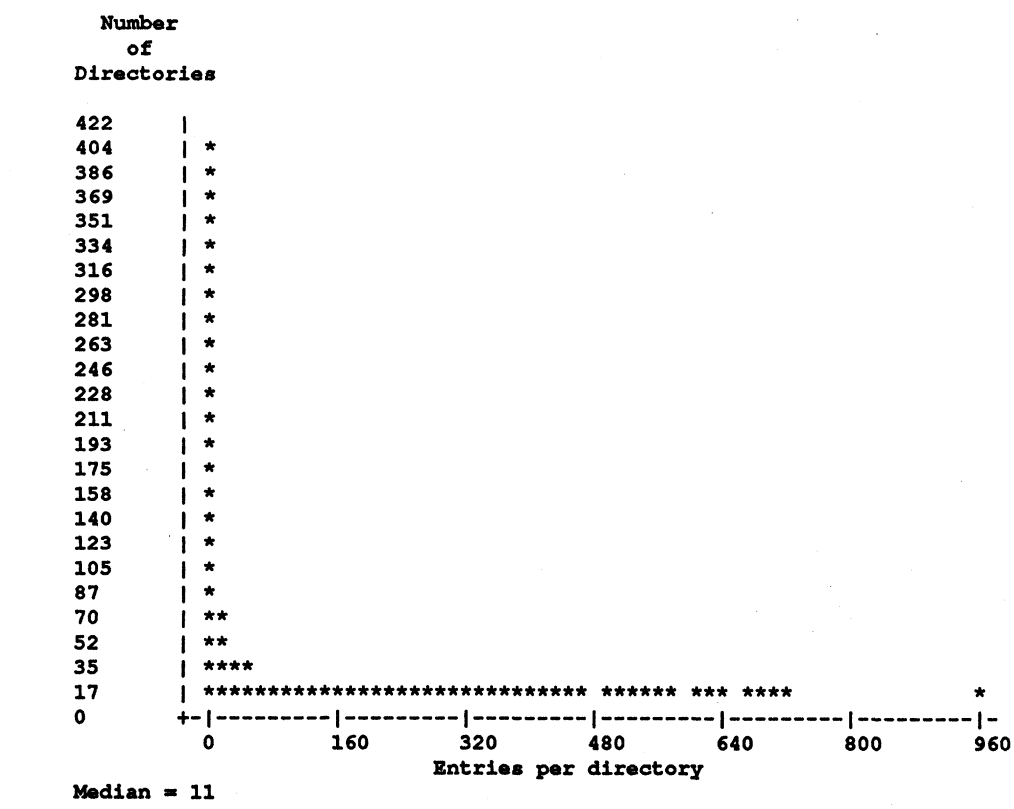
It is clear from the table that most directories are indeed small. The median directory size is 11 entries or fewer, and, except for systems D and E, the mean size is around 20 entries. The value for system D is low probably because many of its users protect their directories so that they could not be listed. The value for system E is high because of its use as an archive for old messages, an application that leads to many large directories.

The skew in distribution towards small directories is especially evident from histograms of directory size. Figure 8-1 shows a histogram for system E. Histograms for the other systems are even more skewed to small directories.

We also broke down the distribution into five classes: user files, Unix kernel and utility sources, Unix utilities (programs, libraries, on-line manual), a large software project, and the "Usenet" bulletin-board system. The results, shown in table 8-4, demonstrate that over a wide range of styles of use, Unix directories tend to be small.

For another perspective on directory sizes, we obtained distributions for several TOPS-20 systems. Because TOPS-20 does not support the subdirectory concept as well as Unix does, users typically have only one directory[3], so the number of entries per directory for TOPS-20 should be larger.

[3]There is no central registry of "users" on a TOPS-20 system; we estimated the number of users on each system by counting the number of top-level directories. This is probably correct to within 10%.



Each vertical bar represents 16 samples on the X axis.

Figure 8-1: Histogram of directory sizes for system E

Use	User files	Unix sources	Unix utilities	Large project	"Usenet" messages
Directories	2723	1106	220	898	275
Depth	9	9	5	6	4
Files	47952	20450	6908	11162	10304
Entries per directory:					
Mean	17.61	18.49	31.40	12.43	37.47
Std. dev.	35.09	27.87	47.16	17.91	67.34
Median	8	9	11	6	11
Maximum	654	250	424	226	482

Table 8-4: Directory size distribution by class of use

To gather the information, we asked TOPS-20 for a directory listing of every file in the file system, and processed the listing to obtain counts of files per directory. The results are shown in table 8-5. The average number of files per user shows little variation from system to system, in contrast to the much larger variation shown in table 8-3 between Unix systems; this may be because disk storage space quotas are more rigidly enforced on TOPS-20 systems. System Z in table 8-5 has an unusual user community, mostly students in introductory programming courses, which accounts for its comparatively small directory sizes.

System	U	V	W	X	Y	Z
Directories	968	461	908	749	1570	5309
Files	41235	17802	31241	30802	60766	75094
Maximum Depth	4	3	3	4	5	3
Users (est.)	678	315	565	526	486	5209
Entries per directory:						
Mean	42.24	38.47	34.19	40.98	38.56	14.14
Std. dev.	87.86	55.86	61.02	72.97	85.26	13.61
Median	14	18	16	16	18	11
Minimum	0	0	0	0	0	0
Maximum	1077	537	713	889	2576	229
Directories/User	1.42	1.46	1.60	1.42	3.23	1.02
Files/User	60	56	55	58	125	14

Table 8-5: Distribution of directory sizes on several TOPS-20 systems

8.3. Implementation: Performance measurements

Many dimensions of file system performance interest us: processor time per operation, elapsed time per operation, total operations per unit time, response time for an isolated operation, internal memory requirements, and disk storage requirements. The operations that interest us are writing and reading properties, and searching for property values.

We compared the performance of the PLDIR and PLING implementations (see chapter 7) against 4.2BSD Unix on roughly similar sets of tasks. These tasks are:

Streamed property creation — Create a series of directory entries and their properties.

Streamed property reading — Read all of the properties in a subtree of directory entries.

Repeated single-property writing — Repeatedly update the value of a single property[4].

Repeated single-property reading — Repeatedly read the value of a single property.

Searching for property values — Determine which entries have a property whose value meets a predicate.

Except for the last task, these require measuring the average times required over a large series of operations, because each individual operation requires so little time that it cannot be measured with meaningful resolution.

8.3.1. Measurement methodology

The performance measurements described in this chapter were obtained by running a suite of test programs. Each test program is meant to highlight one of the tasks listed above. For each program run, we obtained the following performance information:

[4]This task must actually change the value of the property each time, since PLDIR does not update the database if the value is unchanged.

- User-mode processor time
- Supervisor-mode processor time
- Elapsed time
- Main memory occupancy (average and maximum)

In addition, we obtained:

- Counts of disk input and output operations
- Counts of message receive and send operations
- Counts of page faults and context switches

which are helpful in analyzing the performance measurements.

Since these programs were run on a timesharing system, the load on the system affects the results of the tests, especially since programs that do I/O interact with other system activity in complex ways. It turns out that only the elapsed time and the disk activity counts are significantly affected by the system load.

It would of course be meaningless to present the measurements without accounting in some way for system load. Two approaches are possible:

1. Present best-case runs for each program, on the assumption that these took place during conditions of minimal load, and are thus comparable.

2. Explicitly show the relationship between elapsed time and load, because in a production environment loads may be high.

The first approach has the disadvantage that it may be misleading about the performance of an implementation in a production environment. The second approach has the disadvantage that the presentation of the results becomes far more cluttered. Also, Unix reports a five-minute load average, and since most of the test programs run for much less than five minutes, the correlation between load and elapsed time is hard to measure.

Experiments show that the best-case results are quite repeatable; they are a useful basis for comparing the performance of various implementations. The results presented in this section are best-case measurements, usually over at least 25 runs of each program.

We did linear regressions to determine the relationship between processor load and both elapsed and processor times. For most of the test programs, the regressions supported the hypothesis that elapsed time increases linearly with load, for loads over a moderate range. The inaccuracy in the processor load measurements, however, makes it impossible to rule out an alternative hypothesis: the rise in elapsed time approximates linearity at low loads but is worse than linear at much higher loads.

For example, we ran the PLDIR "streamed property reading" test program over loads ranging from 0.35 to 5.46, and analyzed the effect of load on elapsed and processor times. The results of the linear regressions are summarized in table 8-6. Elapsed time appears to increase linearly with load, but the standard deviation of the error is nearly half as large as the multiplier of the load; that is, the confidence band is quite wide. One cannot reliably predict the elapsed time from the load.

Processor time increases slightly with increasing load. This is because of both higher supervisor-mode overhead and because as the load increases, there are more interrupts, which are charged to whatever process is running when they occur.

Experiment: 505 runs of the "streamed property reading" program.
Variables (all times in seconds):

Load	Five-minute average of length of processor run queue (ranges from 0.35 to 5.46)
ElapsedTlme	Elapsed time per run (minimum 20.0)
CPUTime	Total processor time per run (minimum 18.5)
SuperTime	Supervisor-mode processor time per run
UserTime	User-mode processor time per run

Results of linear regressions (all support linearity hypothesis with significance level 0.01 or better):

ElapsedTime	. 18.3* *Load*	+4.96,	r« 0.888,	a(e)«7.01
CPUTime	« 0.779* *Load*	+18.4,	r- 0.669	
SuperTime	~ 0.430* *Load*	+2.94,	r-0.582	
UserTime	* 0.349 * *Load*	+ 15.5,	r * 0.654	

Table 8-6: Correlations between processor load and various performance measures

8.3.2. PLDIR performance

We selected a small subtree of a Unix file system for use as a sample database. The Unix test programs were run directly against this subtree; the PLDIR test programs were run against a database created to contain exactly the same property information. None of the tests involved the contents of the subject files.

The selected subtree contains sources and binaries for a set of networking programs. The subtree is composed of 767 entries in 58 directories, for an average of 13.5 entries per directory; the largest directory has 53 entries. The property information stored by Unix is converted into 10 properties for each file; entries denoting directories are given 2 additional properties in the PLDIR database. Thus, the database contains about 7780 properties.

The tests in this section were run on four different system configurations, described in table 8-7. By comparing the performance of the test programs on these four configurations, we can isolate the effects of varying the processor speed, disk access speed, and operating system performance. The relative processor speeds were derived from a simple-minded compute-bound benchmark called "puzzle" [Bask79, Hans82], and from a second benchmark (sorting an array of 10000 integers) that is more memory intensive.

The average disk access times are taken from the manufacturer's specifications. Although the *'Eagle" (Fujitsu M2351) disk drive has a lower average access time than the RA81 (manufactured by Digital Equipment Corporation), this is not the only difference between the two types of disk. The Eagles are attached to the processor via a relatively simple interface, while the RA81s are attached via a special processor that performs seek operations without CPU intervention. It is not clear if this saves Unix anything; the added complexity of communicating with the special processor may actually increase the CPU time requirements.

System V4 differs from the other three in that it is running a newer version of the Unix operating system, on which a variety of performance improvements have been made [McKu85]. Significantly, the new system caches recent file name translations, which reduces the cost of *statQ* and similar operations.

The first set of tests compares Unix and PLDIR performance; the results of this comparison are shown in table 8-8. The tests in this table were run on system VI, a VAX-11/750 processor with RA81 disks running 4.2BSD Unix. In these tests, no inverted indices were maintained by PLDIR, because Unix does not support indices.

System name	Operating system	Processor	Relative CPU speed (puzzle)	Relative CPU speed (sort)	Disk
V1	4.2BSD	VAX-11/750	1	1	RA81
V2	4.2BSD	VAX-11/750	1	1	Eagle
V3	4.2BSD	VAX-11/780	2.0	1.6	RA81
V4	4.3BSD	VAX-11/780	2.0	1.6	RA81

System Configurations

Disk	Attachment	Average seek time	Average latency time	Average access time	Maximum transfer bandwidth
RA81	UNIBUS/UDA50	28 mSec.	8 mSec.	36 mSec.	2.2 Mb/Sec.
Eagle (Fujitsu M2351)	MASSBUS	18	7.6	25.6	1.86

Disk Drive Types

Table 8-7: Test system configurations

Task	Total number of properties	CPU time	Elapsed time	Properties per second	Speed relative to Unix
Streamed property creation					
PLDIR	7778	138 Sec.	176 Sec.	44	0.34
PLDIR*	7778	95	127	61	0.46
Unix	7778	15.1	59	131	1
Streamed property reading					
PLDIR	7778	18.3	20	389	0.35
Unix	7778	6.8	7	1111	1
Repetitive property reading					
PLDIR	10000	16.3	16.3	613	0.37
Unix†	10000	5.9	6	1666	1
Unix‡	1000	33.4	35	28	0.017
Repetitive property writing					
PLDIR	10000	89.8	93	107	0.075
PLDIR*	10000	31.9	32	312	0.22
Unix†	10000	7.1	7	1428	1
Unix‡	1000	33.5	36	28	0.019

* without incremental logging of changes
† operations by open file ID
‡ operations by file name

Table 8-8: Relative performance of PLDIR and 4.2BSD Unix

The columns in this table, and subsequent tables, are:

Task The aspect of performance highlighted (e.g., 'Streamed property creation") and whether PLDIR or Unix was used.

Total number of properties
The number of file properties written or read by the task.

CPU time
The processor time, including both user-mode and kernel-mode time, required for the task; this is a minimum over many trials.

Elapsed time
The elapsed time required for the task; this is a minimum over many trials.

Properties per second
The number of file properties written or read per second of elapsed time.

Speed relative to Unix
The relative speed of the task, in terms of elapsed time, compared to Unix.

The PLDIR system normally logs individual property updates to a disk file; for property-writing tasks, times are shown both with and without logging. Unix can read or modify the properties of a file either by specifying its name, or by opening the file and using the open file identifier. The latter is much faster for repetitive access to the same property; times for both methods are shown.

The measurements in table 8-8 indicate that PLDIR is about a factor of three slower than Unix for streamed property operations and for repetitive property reading. For repetitive property writing, PLDIR is more than ten times slower than operating Unix on a Unix file identifier.

Actually, this comparison is unfair to PLDIR, since PLDIR normally logs each update to a disk file, while Unix merely changes a value in a resident data structure. If PLDIR is told not to log updates, making it no more crash-proof than Unix, its performance on repetitive property writing improves to a fifth the speed of Unix.

The speed with which a property-based system can repeatedly update a property is crucial to the performance of a file system, because file operations often implicitly cause property updates. However, these updates are done only to intrinsic properties, and it is therefore possible to bypass most of the mechanism for translating these property names, and so improve performance. For example, the file system might maintain a cache of pointers to the values of intrinsic properties for each open file.

The next set of tests demonstrates the effect of increased processor speed; the results of this comparison are shown in table 8-9. The tests in this table were run on system V3, a VAX-11/780 processor with RA81 disks running 4.2BSD Unix.

The rightmost column in the table shows the relative speedup in elapsed time over a VAX-11/750 with the same disk and operating system. The speedup ranges up to a factor of about 2, but in general is closer to the 1.6 factor derived from the sort benchmark (see table 8-7) for the CPU-intensive tasks, and is close to unity for the disk-I/O intensive tasks. This is unsurprising.

The third set of tests demonstrates the effect of different disk drive speed; the results of this comparison are shown in table 8-10. The tests in this table were run on system V2, a VAX-11/750 processor with Eagle disks running 4.2BSD Unix.

Task	Total number of properties	CPU time	Elapsed time	Properties per second	Speed relative to Unix	Speed relative to VAX-11/750
Streamed property creation						
PLDIR	7778	91.5 Sec.	123 Sec.	63	0.42	1.0
PLDIR*	7778	59.7	88	88	0.59	2
Unix	7778	11.9	52	150	1	1.1
Streamed property reading						
PLDIR	7778	10.3	12	648	0.33	1.7
Unix	7778	3.9	4	1945	1	1.75
Repetitive **property reading**						
PLDIR	10000	8.8	9	1111	0.39	1.8
Unixf	10000	3.5	3.5	2857	1	1.7
Unix*	1000	20.5	21	48	0.017	1.7
Repetitive **property writing**						
PLDIR	10000	63.8	71	141	0.052	1.3
PLDIR*	10000	17.9	18	556	0.21	1.8
Unixt	10000	3.7	3.7	2703	1	1.9
Unix*	1000	20.7	22	45	0.017	1.6

* without incremental logging of changes
t operations by open file ID
t operations by file name

Table 8-9: Effect of processor speed on performance: Relative performance on VAX-11/780

Task	Total number of properties	CPU time	Elapsed time	Properties per second	Speed relative to Unix	Speed relative to RA81
Streamed property creation						
PLDIR	7778	137.5 Sec.	170 Sec.	46	0.34	1.0
PLDIR*	7778	91.5	119	65	0.48	1.1
Unix	7778	13.1	57	136	1	1.0
Streamed property reading						
PLDIR	7778	18.3	20	389	0.35	1
Unix	7778	6.7	7	1111	1	1.0
Repetitive property reading						
PLDIR	10000	16.6	17	588	0.38	1
Unixt	10000	6.5	6.5	1538	1	0.9
Unix*	1000	22.0	22.1	45	0.029	1.6
Repetitive property writing						
PLDIR	10000	94.8	99	101	0.072	0.9
PLDIR*	10000	32.7	33	303	0.22	1.0
Unixt	10000	7.1	7.1	1408	1	1.0
Unix*	1000	23.2	24	42	0.030	1.5

* without incremental logging of changes
t operations by open file ID
i operations by file name

Table 8-10: Effect of disk type on performance: Relative performance with "Eagle" disk drive

The rightmost column in the table shows the relative speedup in elapsed time over an otherwise identical system with an RA81 disk. There is almost no performance difference except for the two Unix programs that access file properties by file name. In both cases, the processor time (and hence elapsed time) is significantly reduced when using the Eagle disk drive. This may reflect an anomaly of the particular system used to run the tests, the cost of managing the UDA50, or a bug in the UDA50 device driver. We believe that the speed of the disk drives, within reasonable limits, is not important.

The final set of tests demonstrates the effect of an improved operating system; the results of this comparison are shown in table 8-11. The tests in this table were run on system V4, a VAX-11/780 processor with RA81 disks running 4.3BSD Unix.

Task	Total number of properties	CPU time	Elapsed time	Properties per second	Speed relative to Unix	Speed relative to 4.2BSD
Streamed property creation						
PLDIR	7778	84.7 Sec.	117Sec.	66	0.43	1.1
PLDIR*	7778	55.6	84	92	0.60	1.0
Unix	7778	10.2	50	156	1	1.0
Streamed **property reading**						
PLDIR	7778	10.2	12	648	0.25	1
Unix	7778	3.7	3	2593	1	1.3
Repetitive property reading						
PLDIR	10000	8.9	9	1111	0.4	1
Unixt	10000	3.6	3.6	2778	1	1.0
Unix*	1000	4	4	250	0.09	5.3
Repetitive property writing						
PLDIR	10000	55.3	61	164	0.049	1.2
PLDIR*	10000	18	19	526	0.16	0.9
Unixt	10000	3	3	3333	1	1.2
Unix‡	1000	4.5	4.5	222	0.067	5.0

* without incremental logging of changes
t operations by open file ID
$ operations by file name

Table 8-11: Effect of operating system on performance: Relative performance on 4.3BSD Unix

The rightmost column in the table shows the relative speedup in elapsed time over a system with 4.2BSD Unix running on identical hardware. The only dramatic effect is the factor-of-five improvement for the Unix programs that access properties by file name; this is certainly the result of the name translation cache mentioned earlier. The PLDIR implementation is tuned for 4.2BSD; none of its parameters were adjusted for the tests on 4.3BSD, and it is conceivable that some improvement is possible.

To summarize these comparisons between PLDIR and Unix performance on basic property operations:

- **PLDIR is about three times slower, in terms of elapsed time, than Unix.**
- **The performance ratio is not really affected by varying the processor speed, and probably not affected by varying the disk speed.**
- **PLDIR performance on property updates is potentially problematical for implicit updates to intrinsic properties; special-purpose code may be necessary.**

8.3.2.1. Inverted index performance

The inverted index implementation of PLDIR may not be the most efficient possible. While searches are executed reasonably fast, updates to indices are expensive. As a result, the use of indices reduces performance on property-write operations. This additional cost, and the potential cost of rebuilding inverted indices after a catastrophe, is only warranted if indices significantly reduce the cost of property-based searches.

In table 8-12, we show the effect of maintaining indices on the costs of both streamed property creation and repetitive property writing. The measurements were made on system VI, a VAX-11/750 with RA81 disks, running 4.2BSD Unix. The cost increase for repetitive update of a single property depends only on whether an index exists for that property; it does not depend on the existence of other indices. The cost of streamed property creation, which involves several different properties for each directory entry, does depend on the number of indices that must be maintained.

Number of indices	Updates logged	Number of properties Total	Indexed	CPU time	Elapsed time	Properties per second	Relative speed*
1	yes	7778	765	142.7	179	43	0.98
1	no	7778	765	100.5	134	58	0.95
2	yes	7778	1530	147.9	188	41	0.94
2	no	7778	1530	105.3	141	55	0.90
5	yes	7778	3825	163.1	208	37	0.84
5	no	7778	3825	121.3	164	47	0.77

* speed compared to when no index is maintained

Streamed property creation

Updates logged	Number of properties	CPU time	Elapsed time	Properties per second	Relative speed	Relative to Unix
yes	10000	127.0	131	76	0.71	0.053
no	10000	67.1	68	147	0.47	0.10

Repetitive property writing

Table 8-12: Performance cost of maintaining inverted indices

Table 8-12 shows that the cost of creating index entries is small compared to the cost of creating new directory entries: no more than 5% additional overhead per index. From this, we predict that if all ten properties per entry were indexed, entry creation would be about half as fast as if no indices were maintained. Unfortunately, our pilot implementation thrashes when that many indices are maintained, because of a Unix limitation on the number of open file descriptors per process.

The cost of maintaining index entries is more significant when compared to the cost of updating existing property values. (The cost depends on the distance between the old and new values, since nearly identical keys are likely to reside in the same B-tree disk block.) It may be necessary to defer updates for busy properties, such as "file size," until a file is closed. This would substantially reduce the cost of maintaining indices without significantly reducing their utility.

The cost of rebuilding an inverted index, from the information stored in the property-list database, is surprisingly low. For our sample database, any of the indices can be rebuilt in between 11 and 13 seconds of elapsed time, depending on how the values are distributed. It is actually faster to build an index and search it in the PLDIR system than it is to do the same search twice using Unix, and so it is reasonable to postpone creation of an index until it is needed.

Once an index is created, searches take practically no time. More accurately, a search takes time proportional to the number of "hits" plus the logarithm of the number of index entries. This is much smaller than the cost of searching a Unix file system, which is proportional to the total number of files examined. Even for the sample database of only 767 files, the difference is significant. For example, to find the 25 files between 100 bytes and 200 bytes long, Unix requires 8.4 seconds of processor time and 13 seconds of elapsed time; PLDIR requires 1 second of processor time and 2 seconds overall.

The difference becomes more pronounced when the sample database is a realistic size. For example, we entered into the PLDIR system the "Usenet" message files stored on one system: over 55000 entries in about 260 directories. To find the 77 files between 1025 and 1026 bytes long, Unix requires 2053 seconds of processor time and 2931 seconds of elapsed time; PLDIR requires 2.1 seconds of processor time and 4 seconds overall. If the query instead matches about 25000 entries, or almost half the total database, PLDIR requires 520 processor seconds and 599 elapsed seconds to complete the search. Even if the query matches every entry, PLDIR

completes it in 1016 seconds of processor time and 1180 seconds overall; this is more than twice as fast as Unix, probably because Unix must do a lot of directory name translations.

In summary, PLDIR does appear to require time proportional to the number of hits, about 25 milliseconds per hit, while Unix takes time proportional to the size of the file system. As file systems grow larger, support for efficient property-based searching will determine if such searches are feasible at all.

8.3.2.2. Disk storage requirements

The PLDIR system requires far more storage space for properties than does Unix. While Unix can store in two dozen bytes all the information it will ever know about a file, a property list system may need several hundred bytes — for property names, value strings, and data structure overhead. Conventional systems achieve their space efficiency through positional notation; an extensible one cannot. Further, the PLDIR system stores precomputed information, trading disk storage space for improved response time.

We can calculate the storage space required by Unix, which includes directory files to store file name information, and space used for property information in the *inode* (file descriptor block) of each file. This is not precisely comparable to the space used by PLDIR, because only some of the "properties," and none of the property protections, stored in the PLDIR database are stored explicitly by Unix. However, it is illustrative of the relative storage space requirements.

For the subtree used in the experiment described in section 8.3.2, the 58 Unix directories required 40984 bytes of disk storage, and the inode for each of the 767 files included 28 bytes of property storage space; the total space used by Unix is thus estimated to be 62460 bytes.

When no indices are maintained, PLDIR stores the equivalent property information in 145287 bytes, or about 2.3 times as much space as Unix requires.

For comparison, the 725 non-directory files in this subtree require 2945K bytes of disk space to store their 2553K bytes of content. Fragmentation accounts for the difference of 391K bytes, about what one would expect with a block size of 1K byte. Thus, more than twice the space is lost to fragmentation as is required for PLDIR storage; PLDIR requires only about 5% of the total space allocated to the files.

When PLDIR maintains inverted indices, each index requires about 12% to 17% additional disk storage, over that required without indices. The B-Tree implementation used is somewhat wasteful of storage space, since it does not attempt to guarantee that nodes are more than half full. In fact, they tend to be closer to half-full than full. Too, each B-Tree key is 16 bytes long; a more imaginative coding scheme could reduce this to perhaps 12 bytes.

The indices maintained by the test programs in table 8-12 take between 21K bytes and 27K bytes. Since each stores the same number of keys, the difference can be ascribed to the way in which keys are distributed in the B-Tree nodes.

8.3.2.3. Memory requirements

The memory requirements of the PLDIR system can be estimated from accounting information provided by Unix. PLDIR requires about 200K bytes when indices are not being maintained, and about 350K bytes when indices are maintained. Only about 50K bytes of this is code; the rest is mostly devoted to various caches.

It is not possible to directly compare the memory requirements of the PLDIR system with Unix, because Unix kernel storage allocation cannot easily be ascribed to particular functions.

8.3.3. Performance in a distributed system: vectored operations

We have asserted that the extensible property-list approach is appropriate for building heterogeneous distributed systems. Performance in a distributed system usually suffers as the number of message exchanges increases. With an extensible system, a client not knowing what properties a file had would apparently need to read them one by one; with a non-extensible system, all the properties can be read in a single operation. We must worry about doing an order of magnitude more remote operations if we choose an extensible system.

We can avoid this penalty by collecting the results of an iteration at the server before shipping them *en masse* to the client. We call this method *vectorization* (see section 6.7.4). Vectorization depends on a Remote Procedure Call (RPC) mechanism that supports effectively unbounded message lengths, since a vector may contain many properties.

We apply vectorization to iteration over the properties of a file, and over the set of entry names in a directory (or the set of version identifiers of a file), since these are common operations. We chose not to vectorize property-write operations, since clients usually only update isolated properties.

8.3.3.1. Performance of the underlying RPC system

Before we can evaluate the performance of vectored operations, we must know how efficient the underlying communication mechanism is. The only RPC implementation available to run in the environment where PLDIR was developed is distinctly inefficient: an improved version of the TCP-based implementation of "Courier" for 4.2BSD Unix. TCP [Post81b] is a poor medium for RPC on a local area network, because of its high overhead. In particular, TCP expends a lot of effort to erase packet boundaries, while the RPC layer expends a lot of effort to deduce message boundaries. Nelson [Nels81a] discusses this problem in more detail.

Table 8-13 summarizes the performance of this RPC implementation for remote execution of functions with varying numbers of arguments and return values. The tests were run on VAX-11/750s, connected by a 10M bit/sec Ethernet. Each test was run in two configurations: with the client and server processes running on the same processor, and with each running on a separate processor.

Number of Arguments passed	Value Returned?	Elapsed time per call
Client and server on same processor:		
0	no	23 milliseconds
0	yes	47
1	no	34
1	yes	58
Client and server on different processors:		
0	no	14
0	yes	41
1	no	13
1	yes	47

Table 8-13: Performance of 4.2BSD/Courier Remote Procedure Call implementation

Essentially all of the elapsed time can be ascribed to processor time. Elapsed time improves when the client and server are on different processors and can overlap execution of the TCP code; the RPC and higher level code is entirely synchronous and so cannot be overlapped. The number of messages does not depend on the number of arguments or return values; with an efficient RPC implementation, all four types of function call should require approximately the same amount of elapsed time. In fact, Birrell and Nelson [Birr84] for their RPC implementation show elapsed times for these functions of about 1 millisecond.

Since the RPC we used is an order of magnitude slower than is demonstrably possible, our measurements of remote property operation costs should not inspire as much panic as they would if taken at face value.

8.3.3.2. Relative performance of vectored operations

We compared the performance of vectored and non-vectored remote operations to that of the same operations performed locally, for streamed property reading and listing all the entry names in a directory. We also compared the (non-vectored) performance of repetitive remote property reads and writes to that of the corresponding local operations. All tests were run on a pair of machines each configured as system V1: a VAX-11/750 with RA81 disk drives, running 4.2BSD Unix. The processors were connected by a 10M bit/sec Ethernet.

Task/Vectored	Properties	Processor time Client	Server	Elapsed time	Messages	Properties per second	Speed relative to local
Streamed Read/No	7778	151 sec	209 sec	464 sec	20476	17	0.043
Streamed Read/Yes	7778	23.8	48.0	84	1886	93	0.24
Repetitive Read/No	10000	141	201	443	20006	23	0.037
Repetitive Write/No	10000	142	196	437	20008	23	0.073
List Names/No	767*	27.7	40.4	86	3878	9*	0.081
List Names/Yes	767*	13.7	23.7	47	1886	16*	0.15

*Entry names

Table 8-14: Relative Performance of Vectored Remote Operations

The results are shown in table 8-14. Vectorization makes a real difference; a factor of 4 to 5 in both processor and elapsed time for streamed property reading, and a factor of about 2 for directory name listing.

The "Messages" column in the table show the total number of messages exchanged; each remote procedure call requires two messages. For all but one of the tasks in the table, the message rate is within 10% of 44 messages per second, or 45 milliseconds per remote call. This is about what we achieved for null operations (see table 8-13) and indicates that performance is limited by the cost of RPC.

We can estimate the performance with an RPC that is ten times faster; this is certainly attainable on hardware similar to ours [Birr84]. In this case, on a local-area network the non-vectored operations should be about 20% slower than the vectored operations, and about a factor of three slower than local operations. Such an RPC implementation is not now available, nor is it likely that we can achieve such performance over long-haul networks[5], so vectored operations are necessary to achieve good performance.

8.3.4. PLING performance

Because PLING performance is two to three orders of magnitude slower than PLDIR, it is pointless to try to quantify it exactly. It would have taken too long to measure its performance on tasks complex enough to be interesting.

---

[5]Since RPC is synchronous, the speed of light is an inherent limit on elapsed time. For example, if the client and server are on opposite coasts of the United States, the minimum possible round-trip time is about 30 milliseconds. Via a geosynchronous-satellite channel, the minimum is about 500 milliseconds.

8.3.4.1. Performance of INGRES/EQUEL

As noted in section 7.5.3, the "free INGRES" relational database system, especially when accessed from a program via EQUEL, is quite slow. To get some idea of just how slow the combination is, we wrote a test program to time operations on a simple database. The database consists of a single relation, whose tuples are made up of an integer key and a 32-byte string. The results of this experiment are summarized in table 8-15; the program was run on a lightly loaded VAX-11/750 running 4.2BSD Unix. Only elapsed times are given; accurate processor times could not be obtained, but the program is essentially CPU-bound.

Task	Elapsed time for 100 repetitions	Operations per second
Null operation	8 seconds	12.5
Insert tuple	67 seconds	1.5
Read one tuple	58 seconds	1.7

Table 8-15: Performance of INGRES/EQUEL

The implication of this experiment is that this database system limits us to at most two database operations per second; the PLDIR system performs at least two hundred operations per second under similar conditions.

We then ran the "streamed property creation" and "streamed property reading" tasks on two tiny sample databases, using both PLING and PLDIR. The first sample database consisted of one directory containing only 19 files with a total of 185 properties; the second contained 69 files with 684 properties. Strings were limited to at most 31 characters; INGRES stores only constant-length strings and its performance degrades substantially when longer strings are used.

The performance of the two database implementations is compared in table 8-16. Since the INGRES database runs in a separate process, the processor time figure for a PLING task is the sum of the processor times for the PLING program and the INGRES program; the latter was hard to measure accurately, and may be slightly low.

Task	Number of files/properties	Processor time	Elapsed time	Properties per second	Speed relative to PLDIR
Streamed property creation					
PLING	19 / 185	425 Sec.	521 Sec.	0.36	0.01
PLDIR	19 / 185	3.4	5	37	1.0
PLING	69 / 684	2089	3127	0.22	0.005
PLDIR	69 / 684	11.5	14	48	1.0
Streamed property reading					
PLING	19 / 185	95	144	1.3	0.007
PLDIR	19 / 185	0.8	1	185	1.0
PLING	69 / 684	541	758	0.90	0.004
PLDIR	69 / 684	2.0	3	228	1.0

Table 8-16: Performance of PLING relative to PLDIR

These results indicate that PLING is indeed at least two orders of magnitude slower than either PLDIR or Unix. These measurements cannot be extrapolated to more realistic database sizes; note that while PLDIR performance seems to improve as the database gets larger, because of economies of scale, PLING performance gets worse, perhaps because of inappropriate algorithms or data structures. They do indicate, though, that to match PLDIR performance using a general-purpose database system, such a system would have to perform several hundred transactions per second.

8-3A2. Disk storage requirements

PLING uses far more disk storage than does PLDIR. For the sample database of 19 files with 185 properties, PLDIR used 4236 bytes of disk storage; INGRES required 97K bytes. For the sample database of 69 files with 684 properties, PLDIR used 13106 bytes of disk storage; INGRES required 105K bytes. There are a number of explanations for the difference:

- INGRES appears to preallocate disk space in large chunks; this is why it does relatively better with the larger database.
- The INGRES database is meant to support general database applications, and thus maintains more administrative information.
- INGRES does not support variable-length strings, and hence allocates a lot of unused storage space for each string. The use of string values larger than 31 characters would substantially increase the storage space requirements.

We did not have the patience to load larger sample databases into PLING to see if INGRES's use of disk space improved significantly with increasing scale; we do know that PLDIR's disk storage requirements scale linearly.

8.4. Use of alternative approaches

In section 8.3.2, we examined the performance of the PLDIR system, and pronounced it acceptable, but we do not take for granted the desirability of a special-purpose, universally available property database system. We could use either a more general approach, by employing a general-purpose database system, or we could use more specific approaches, solving particular problems with application-specific databases. In this section, we argue that a general-purpose database is reasonable, perhaps even preferable, but only if its performance is acceptable.

8.4.1. Why not use a general-purpose database system?

The poor performance of the PLING implementation is clearly not representative of what a high-quality database system could provide, and so we are unable to exclude the use of a general-purpose database-management system (DBMS) for storing file properties. There are several benefits to using a DBMS:

- A high-quality DBMS *might* perform better than a special-purpose system, particularly if significant operating system support is provided for the DBMS.
- A DBMS, especially a relational DBMS, used as the basis for a file-property or directory-property system, can provide more functionality than the PLDIR system. For example, queries could involve joins, and more complex selection predicates.
- Since a DBMS is increasingly expected as a component of a modern computing environment, it makes sense to integrate file-property support with a DBMS already used for other purposes. This avoids duplication of effort, obviates the need for users to learn more than one query language, and allows clients to combine property information with other DBMS data.

If a DBMS system could be made to perform well enough, then these for reasons we should prefer it to a special-purpose system such as PLDIR.

Can a DBMS system perform well enough? As we saw in section 8.1, property operations are relatively frequent in proportion to other file system operations, so poor performance on property operations would significantly harm overall file system performance. A file system does implicit property operations, on intrinsic properties, during all data transfer operations, so a cleanly-layered DBMS-based property system might become a significant bottleneck. A special-purpose system, designed with specific understanding of the problem domain, has the advantage that it can include special paths for handling these implicit operations, perhaps even special data structures, and can thus avoid this bottleneck.

We have concentrated on the PLDIR implementation and its performance not so much as a description of the best way to implement properties, but as an existence proof that techniques exist that allow us to obtain acceptable performance in an extensible property system. If a DBMS with comparable performance were available, it would probably be the better choice. Benchmarks of existing, commercially available relational database systems show transaction rates of between 1 and 20 simple updates per second, on hardware similar to ours [Bitt83, DeWi85, Hawt85]. Crude projection of the performance of PLING implies that even the best relational database systems are still an order of magnitude slower than PLDIR for simple operations; they may be faster for complex queries.

8.4.2. Why not use application-only databases?

When a DBMS is not available or convenient to use, applications often make use of private database systems to solve their specific problems. For example, various systems (including Unix and VAX/VMS) maintain a simple database for user authorization. If one examines a 4.2BSD Unix system, for example, one finds many simple application-specific databases, including:

`/etc/passwd`	User authorization
`/usr/spool/at`	Tasks to be performed at specific future times
`/usr/adm/wtmp`	Accounting records
`/usr/lib/aliases`	Mail forwarding and mailing list membership

Each of these databases is implemented differently: the first is a text file with one line per user, the second a directory containing one file per task, the third a binary file containing one record per terminal session; the last is built on an extensible-hash-table package that is part of the Unix library and so is widely used. Although any of these databases could have been implemented using any of the four techniques, in most cases the implementation chosen is appropriate.

Why not follow this paradigm of using application-specific databases to solve the problems we propose solving with extensible file properties? Each application would get the "best" implementation, and the file system would not be cluttered with the significant additional mechanism needed for full support of file properties[6].

The problem with this line of argument is that the best solution for a problem viewed in isolation is not always the best, or even a good, solution to the problem when it is viewed in the context of a larger system. Application-specific databases, as an alternative to a property-based file system, have the following problems:

- A file property database is needed in any event to support general file system func-

[6]This is essentially the argument that experienced Unix programmers make when it is proposed that Unix be extended to include file properties. We do not make such a proposal, but for different reasons.

tions. These are functions that cannot be considered to be part of any specific application.

- **A database implementation that supports sophisticated queries is hard to implement correctly and efficiently. Rather than each application re-implementing this mechanism, they should all share a single, well-engineered implementation.**
- **When each database has a different structure, one cannot create a single tool capable of manipulating similar information in multiple applications; for example, to do keyword-based searches for both bulletin-board messages and online-manuals. Nor can one easily combine information from several applications, for example to find the personnel record for the sender of a message. To do either of these requires a uniform database interface, if not a logically centralized database.**

Of course, application-specific databases (including all four examples listed earlier) often store information that is not reasonable to store as file properties. This is not an argument for keeping them at the expense of a property-based file system; rather, it is an argument for use of a general-purpose DBMS for both property and non-property data.

Part III

Prospects

Introduction to Part III

In part II we presented our solutions to the problems we hinted at in chapter 2. We now proceed to show how our solutions can be applied to a variety of specific problems, why our solutions are better than what has been available, and what problems remain to be solved.

We start, in chapter 9, by looking at several problems arising in current systems for which the traditional solutions are woefully inefficient. Our file system proves dramatically faster.

We then look more generally at an interesting problem that demands adequate file property support. In chapter 10, we see how file sharing in a heterogeneous distributed system requires both uniform and extensible property support.

We do not have the temerity to suggest that we have solved all the problems related to file properties. In chapter 11, we point out some of the remaining problems, and suggest approaches to their resolutions.

Chapter 9

Better solutions through File Properties

In this chapter, we look at how the property-based file system design of chapter 6 can provide more efficient or more reliable solutions to real problems arising in existing systems. These can be separated into two categories: "search[99] problems, in which a user or system administrator must find a set of files based on some predicate, and "knowledge" problems, in which some piece of knowledge about a given file must be available in order to handle it properly.

In general, our design is an improvement over traditional file systems because it makes searching more efficient, and because it makes it easier to store important information about files. The distinction is not firm; efficiency and convenience are intimately related, since almost any problem can be solved with enough effort. What we would like to do is to reduce the cost and complexity of solutions.

We draw these examples from Unix systems since they are widely used and familiar in our environment. One should be able to generalize these examples to other systems.

9.1. Problems involving search

When a user searches for a file based on its properties, several factors affect the cost of the search:

- **Scope:** What set of files may potentially match the predicate? The scope of a search might be the versions of one file, the entries in a directory, an entire file system, or multiple instances of any of these.
- **Frequency:** How often is the search performed? Certain administrative functions may be performed only once a day; keyword-based information retrieval might be done continually.
- **Availability of information:** Is the information tested by the predicate explicitly available as properties, or must be it be extracted from the file data?
- **Organization of databases:** Is the information tested by the predicate organized for rapid retrieval, or is it dispersed in such a way as to require linear search?

Scope, frequency, and (to some extent) availability depend on the particular problem being solved. Availability and organization depend on the design of the file system; extensible property support improves availability for non-intrinsic properties, and advanced data structures, such as inverted indices, improve organization for broad-scope searches.

Improved availability and organization are subject to cost/benefit tradeoffs: they come at the expense of storage space and response time for non-search operations. Our file system design better solves the problems described in this section because the benefits outweigh the costs.

All these problems have solutions within the traditional Unix framework. We are not providing solutions to unsolved problems; we are providing *better* solutions. Almost any solution will work, given enough time; the traditional Unix approaches often take more time than is available, and so are not always practical.

9.1.1. USENET message expiration

USENET, sometimes referred to as "netnews," is a decentralized distributed database, covering several thousand sites, for the exchange of messages in a "bulletin board" manner. USENET maintains several hundred "news groups," each a bulletin board concerned with a specific topic (such as local area networks, programming language standards, cooking, etc.). Each site stores a copy of most recent messages; this allows a user to read messages without requiring network transactions. Database updates are batched, often over slow network links that would not support interactive use.

The Unix implementation of USENET stores each message as an individual file; this allows the use of many standard Unix tools and file system operations to manipulate messages, instead of requiring a special-purpose text-retrieval database system. Message files are grouped into Unix directories by topic; all the messages in a single news group are stored in a single Unix directory.

Because of the immense volume, more than 2000 messages each week, it is necessary to delete expired messages. Unless an expiration date is explicitly specified in the message, a typical policy is to delete any message more than two weeks old. Unfortunately, this is an expensive task.

The solution adopted under Unix has been to scan all stored messages, parsing their headers to find either an explicit expiration date, or the date of origination, to determine if they should be removed. This is extremely expensive, requiring processor and elapsed time measured in hours. An alternative solution is to use the Unix *find* program to locate and remove the message files created before a certain date; this is somewhat faster, but is technically incorrect because it ignores explicit expiration dates that might be intended to preserve an important message.

The fundamental problem with these solutions is that they require examining each of the stored messages; this means that if the expiration policy is changed to retain messages for a longer period, the cost of removing expired messages will go up even though the number of messages deleted per week stays constant. Both solutions involve traversing a tree of path names; this is expensive in Unix because it means performing a path-name lookup on every file. Also, neither method has available to it as file properties the information it needs to make the expiration decision; the "official" method requires the costly reparsing of every file, while the other method accepts the possibility that some messages will be handled incorrectly.

An efficient, yet correct, solution to the USENET expiration problem must be able to select messages for deletion without examining any more data than necessary. One way to do this is to maintain a database with one record per message, organized so that the records with a given range of expiration dates can be retrieved rapidly. The USENET system could implement such a database[1]. However, this would not be necessary with our file system design.

We propose this solution: as the USENET software receives each message, it parses the

---

[1]A later version of the USENET software does use an application-specific database. Its performance is still poor, probably because the database implementation is inadequate.

headers to determine where to file it. At this point, either an explicit expiration date or the origination date is available, and should be stored as a property value in the directory entry for the message file. The system should maintain an index on this property; thus, when it is necessary to delete stale messages, the expiration program can perform a "global query" for files whose explicit expiration dates lie in the past, and another query for files whose origination dates lie too far in the past. If one wishes to implement different expiration policies for different news groups, this can be done using a "join" operation, or by performing the origination-date queries over the specific directories involved.

Note that these properties are stored in the directory entry, to allow rapid searching over a large collection of files. There is no cache-consistency problem, however, because we know that these values cannot change once they have been stored; USENET messages are immutable.

How do the methods compare in cost? We used a typical USENET database of about 9500 messages, 1950 of which were more than 2 weeks old and therefore ripe for expiration; the host was a VAX-11/750. The USENET *expire* program took just short of an hour to find all the stale messages, delete them, and update some database information; most of this time is spent parsing messages. By comparison, the Unix *find* command took about 8 minutes to find all the messages more than two weeks old. It took under a minute for PLDIR to locate the same set of messages using an index, and just over a minute for PLDIR to find the messages without using an index.

It would therefore seem that the use of an index does not buy very much. However, because *expire* is so costly, it is only run once a week. Running *expire*, *find*, or PLDIR without using an index once a day would mean expending roughly seven times as much effort. However, if PLDIR using an index were run daily, because it requires time proportional to the number of messages to be deleted, this would cost in total about the same as running it once each week.

In summary, PLDIR provides the best performance by an order of magnitude, with or without an index. If an index is maintained, the database can be trimmed more often and thus would require perhaps 30% less disk space.

9.1.2. USENET keyword searching

Another problem posed by USENET, or any large database of text, is how to find the interesting entries. Most USENET readers read every message with a "Subject" line that appeals to them, perhaps saving (in another file) those particular messages that prove interesting. This does not help if one realizes only several weeks later that a message was interesting, or if one doesn't read the relevant messages.

One approach to document retrieval from large text databases is to assign a small set of keywords to each entry; users specify queries as combinations of keywords. The appropriate choice of keywords is difficult; one case study found "retrieval effectiveness to be surprisingly poor" [Blai85]. Still, the composers of messages might learn to assign useful keywords; they already tend to assign meaningful "Subject" lines.

USENET allows the composer of a message to attach a "Keywords" line, but does not provide a keyword-based retrieval mechanism; there is no efficient way of finding messages based on keywords. Thus, the assignment of keywords is spotty and often frivolous.

As an experiment, we began with a large USENET database, containing about 50000 messages, on a system where an excess of disk space makes it possible to retain messages for a longer time; essentially all of the measurements in this section scale about linearly with the database size. We ran a program that parsed both the "Subject" and "Keywords" lines of

every message, eliminated noise words such as "the", and used the remaining words to create property names in the PLDIR database[2]. For example, a message whose subject is "Easy Chicken and Walnut (Fesendjan) Stew Recipe" would be assigned the boolean-valued properties "keyword-easy", "keyword-chicken", "keyword-walnut", "keyword-fesendjan", "keyword-stew", and "keyword-recipe". A user might later, for example, ask for the files with keywords "chicken" and "recipe", and be shown this message. Assigning keyword properties to each message increases PLDIR's storage space requirements by about 50%.

If the user can focus the search onto a small set of newsgroups, then there is no need to maintain inverted indices. For example, on a system where the "cooks" newsgroup contains over 1500 messages, it takes about 5 seconds for the PLDIR system to find all the messages with the "keyword-chicken" property. To find the same messages by searching the text of the files for the string "chicken" occurring in a "Subject" line takes 370 seconds.

If the search cannot be focused on a small set of newsgroups, when is it reasonable to create an inverted index? One extreme would be to create an index for every keyword; there would be several thousand such indices, but most would be quite small and would not need to be constructed as B-Trees. However, most of these indices would never be used, and the space overhead might be significant.

The other extreme would be to never create an index on a keyword; PLDIR can search a large set of directories for a given property much faster than a textual search could search the corresponding files for a keyword. It took 200 seconds to search the database of 50000 messages for all occurrences of the "keyword-chicken" property, without using an inverted index.

If a global index is needed, it takes about twice as much time to create one as it does to simply do a global search for messages with that property. (This is actually slightly less time than it took to do the text search for "chicken" over one newsgroup.) This means that if one expects to do more than a couple of queries for a particular keyword, an index should be created. Otherwise, there is not much point in creating an index; the directory-local search mechanism is sufficiently fast.

Of course, the advantage of a textual search is that it can locate messages based on words or phrases appearing anywhere in their texts, not just in their headers. This mechanism is always available, in the event that the more efficient property-based search fails to discover enough matching messages, or if it is necessary to find all relevant messages. It is also terribly expensive; to search the 50000-message database for all occurrences of the work "chicken" took more than 3 *hours*, and merely to search the 1500 messages in the "cooks" newsgroup took 279 seconds.

In summary, it is clear that if the user can limit the scope of a keyword-based search to a single newsgroup, then the PLDIR system can provide answers almost instantaneously. If a larger scope must be searched, then PLDIR can provide about an order of magnitude better performance than simply doing textual searches.

The philosophy of the Unix USENET implementation, and of Unix in general, is to use standard Unix facilities instead of special-purpose code, whenever possible. One could certainly implement a special-purpose database to support USENET message expiration and keyword-based retrieval, but our property-based file system design provides sufficient mechanism to make this unnecessary.

[2]Our keyword-assignment scheme is crude; it should recognize such forms as plurals and tenses, and reduce them to their root words.

9.1.3. File expiration

We can generalize the USENET message-expiration problem to a general problem for files: some files have finite, predictable lifetimes, and need not be retained forever. In the Unix world, such files include editor backups, *'deleted" mail messages, core dumps, etc. The typical solution is to run, once a night, a process to search the entire file system for files whose names match specific patterns (e.g., ending in . BAK) and which haven't been accessed within a specific period.

There are two serious problems with this method:

1. The decision to expire a file is based on the file name and a globally-enforced lifetime constant, not the intention of the file's creator.

2. The search for expired files is relatively expensive, especially since most files will never expire. On a Unix system with about 43600 files, it took about 20 minutes to find the editor backup and checkpoint files.

We can do much better by using the index facilities of our property-based file system. The second problem is trivially solved by the use of an index; the former problem requires the use of new properties to communicate intentions between client and file system.

First, the decision must not be based on a file name. Any utility that creates temporary files, such as an editor that stores checkpoints, can easily mark them with an expiration property.

Second, a client might specify an expiration date in several ways:

- Absolute time: after a specific date, the file may be deleted.
- Relative time: after a specified interval following the last use of the file, it may be deleted.
- When necessary: if the file system needs space, it can delete any file with this property (although it should prefer the less recently used), since the client is capable of recreating it if necessary.

Use of absolute-time expiration results in minimal work for the file system, since the cost of finding expired files is directly proportional to their number. However, clients must continually update absolute-time expiration properties, so for most applications this is an inconvenient method. Relative-time expiration is much easier for clients, since the expiration property need be set only once per file. It slightly increases the cost of finding expired files, since the system must first find all the files with a relative-time expiration property, then determine which of these are past their expiration dates. On the balance, relative-time expiration is probably cheaper, and certainly results in better response time.

When-necessaiy expiration has minimal cost; it is useful only if clients are willing to treat some files as "caches," rather than truth. There are actually many such files; for example, object modules can often be recreated from sources and are thus only retained to speed recompilation.

This is one application where an application-specific database is clearly not the right solution. The "application" is the file system itself, so the database should be managed by the file system. Once we recognize the need to provide efficient property-based search functions as part of the file system, it no longer makes sense to suggest that specific applications should reimplement these functions.

9.1.4. Weekly report of locks held

Multi-programmer software projects require some scheme to prevent two programmers simultaneously editing the same source file. The Revision Control System (RCS) [Tich82] allows a programmer to lock a file while it is being edited. An RCS lock persists until the programmer explicitly releases it; locks do not time out and are not broken by system restarts, since it may take quite a while to make a change. Unfortunately, locks may inadvertently be left set, perhaps by a programmer who leaves work before an update is finished and forgets to complete it later on.

It is thus useful to remind programmers periodically what RCS locks they currently hold. In the Unix implementation, this requires a scan over the entire file system, or perhaps a selected subtree, to locate the locked files. It cannot be done often; for a typical software project with perhaps 14000 potentially lockable files, it takes 30 minutes to locate those that are locked.

If RCS locks were stored as properties with an inverted index, it would take a matter of seconds to find the locks held by a given programmer. This would make it feasible to tell programmers what locks they hold, for example, each time they log in.

9.2. Problems involving additional knowledge

Some problems involve not the location of a particular file, but determination of how to use it once it has been found. Unlike search problems, solutions to these knowledge-dependent problems should improve convenience rather than performance. The ready availability of pertinent information about a file might not significantly speed our handling of the file, but it can enable cleaner, more automated approaches.

In this section, we look at two different examples of knowledge-dependent problems. The first is the management of site-specific files; the second is the use of "type" properties to control proper treatment of files. The common thread is that, in each case, a relatively small amount of application-specific knowledge, attached to a file and readily available, can be combined with minor changes in programs to yield a dramatic improvement in the "automatic" handling of files.

9.2.1. Site-Specific files

In almost any large computer system, certain "system files" will be specific to the given system. These site-specific files might include hardware configuration information, user authorization, mail forwarding databases, site-licensed software, etc.

Site-specific files can cause problems for system managers, especially in an organization where several otherwise identical systems are managed together. For example, if software updates are automatically distributed from one machine to all the others, one must be careful not to distribute the site-specific files. Further, when a new system is installed, perhaps by reloading a file system dump from an existing system, one must be careful to find all the site-specific files and initialize them.

The programmer (or program) who creates a file can easily determine if it is site-specific. With an extensible-property file system, the file should be labeled with a "Site-Specific" property. This allows system managers to avoid the mistake of treating the file as if it were not site-specific:

- **All site-specific files can be located:** A system manager can use property-based search mechanisms to rapidly locate all site-specific files, with confidence that none have been forgotten.
- **Programs can check to ensure that the appropriate file is being used:** A program that reads a configuration file, and that knows that the file is possibly site-specific, can check to make sure that the *right* site-specific file is being used. This is accomplished by setting the value of the "Site-Specific" property to be the name of the site for which the file is appropriate; if the file is inadvertently moved, the mistake is obvious.
- **Distribution systems can avoid disrupting site-specific files:** Automatic software distribution systems (such as the *rdist* program in 4.3BSD Unix) can be trained to refuse to install or update a file marked "Site-specific"; an entire directory sub-tree might also be labeled site-specific.

Thus, with very little additional mechanism, the availability of a rather small piece of knowledge allows us to resolve most of the system-management problems related to site-specific files. We can deal with site-specific files intelligently, and we can apply relatively unsophisticated solutions to site-general files, knowing that these solutions will not inadvertently foul up site-specific files.

9.2.2. Assignment and Interpretation of Type Properties

Users often strive to represent in some way "the type" of a file. In a traditional system, if it is even possible to represent file type, one is limited to assigning a single type to a given file. However, assigning a type to a file really means assigning the file to a class, and files often belong to more than class at a time; it should be possible to assign multiple types to a single file.

For example, take a file which is meant as input to a document compiler, such as Scribe [Reid80]; what is its type? It is a text file, but this may not be sufficiently detailed; one may need to know that it is represented as ASCII, with linefeeds separating lines, and containing no overstruck characters. It is not only a text file: it is also a Scribe input file, and one might need to know to which version of the Scribe "source language" it conforms. If the file is to be run through a spelling checker, one should know the natural language it is written in; it might be important to distinguish between American and British English. If the final document will be given as a paper at an IEEE conference, it could be given a type of "IEEE format paper"[3].

Viewed in a larger context, the file might be assigned to still other classes, and thus have other types. It could be a member of the class "part of my thesis" or the class "paper on Computer Science," with appropriate subclasses that correspond to the keywords used to classify papers.

The range of possible type properties is potentially infinite. How are type properties assigned to a file, and how are these properties interpreted? The answers to these questions depend upon the distinction between human agents and automatic ones; humans are better able to deal with taxonomic distinctions, especially the invention of new classes, than are programs.

---

[3]In fact, this is done for Scribe input files by information embedded in the source file, but this suffers from the disadvantages of embedded properties, discussed in chapter 5.

9.2.2.1. Assignment of types

When a file is created by a program, the program might not always have sufficient semantic understanding to assign the proper types. While, for example, an assembler does know enough to assign its output file the types "object module", "binary", etc., a text editor used to create a paper that touches on Pascal would be hard put to deduce on its own that it should assign a "keyword-Programming_Languages" property to the file.

The more interesting (restrictive) type properties must be assigned based on information originally supplied by humans, although the use of layered abstractions can eliminate a lot of effort. (An example of a layered abstraction is a general-purpose text editor used in "Pascal mode"; the human must specify an intention to edit a Pascal file, to obtain specialized support from the text editor, but then should not later have to intervene to set the file type.)

On the other hand, many broad types, such as "Text", can be assigned automatically. While broad types do not convey the precision of narrower ones, they are useful in many situation, such as deciding what mode to use for a file transfer protocol. For files created as the result of running a program, rather than by direct request by a user, the automatic assignment of types is the only way to ensure that they are assigned consistently and universally.

9.2.2.2. Interpretation of types

While intelligence is necessary to make full use of the syntactic power of file properties, simple-minded automation goes a long way. Certainly for a small set of frequently used types, automatic interpretation is possible. Programs that check the types of their input files can avert careless errors, such as an attempt to edit a binary file with a text editor. Programs can also check the types of pre-existing output files; for example, a user might accidentally ask a compiler to write its object file output on top of an existing text file. A program should refuse to overwrite an existing file with one of an incompatible type.

Inevitably there will be file types that programs do not understand. A reasonable reaction for a program presented with input of an unknown type is to reject it, giving a specific reason. Perhaps the file was presented to the program by accident; such would be obvious to a user if, for example, the print program failed with the message "cannot print files of type 'object module'." In many cases the user might simply overestimate the capabilities of the program to deal with novel file types; when informed of this, a human can apply common-sense intelligence to find a solution.

Programs should handle file types automatically in the most frequent cases, and recognize the few cases where a human must become involved. It is unreasonable to strive for automatic handling in every circumstance; we should not reject a solution simply because it must occasionally depend on human intervention.

Chapter 10
File Properties in Heterogeneous Distributed Systems

One of our primary motivations in rationalizing the treatment of file properties is to lower the barriers to file sharing in heterogeneous distributed systems. We now analyze the particular problems encountered in heterogeneous systems, and show how they can be overcome by careful design.

We must avoid the dangerous temptation to try to solve the heterogeneity problem retroactively and for all cases. Prospective solutions are simple, so long as system designers agree on a compatible method. Retroactive solutions are often impossible; the wrong design decisions have been carved in stone. Without a common approach for communicating file properties, we will be unable to construct heterogeneous distributed systems.

It is instructive to look at the failure of the National Software Works (NSW), one of the first attempts to build a heterogeneous file-sharing system. NSW failed because the systems it linked did not have in common a suitable base of primitive operations; it was attempting to link systems that simply could not be linked. Programming languages such as Pascal are portable between processor architectures because the language can be implemented in terms of primitives that are available on almost any modern processor; file systems do not yet share a sufficient set of primitives. For this reason, we should not expect perfection when sharing files in a heterogeneous system.

In this chapter, we look at three aspects of the use of files in distributed systems: file transfer and remote access, file data format and property translation, and file migration. Each of these involves the use of file properties.

10.1. Interhost use of files

In a distributed system, a process on one host often needs to use a file on a foreign host[1]. There are two methods for doing this: a File Transfer Protocol (FTP) may be used to create a copy of the file on the local host, or a File Access Protocol (FAP) may be used to give the process direct access to the file on the foreign host.

FTPs are much easier to implement, but sometimes one does not want to make a copy: the file may be shared by simultaneous processes on different machines, or only a small part of a large file will be accessed, or there may not be space on the local machine for the file, etc. With either an FTP or a FAP, we should insist that the properties of a file are as accessible as its contents.

---

[1] We will refer to the host on which processing is done as the "local host," and the one on which the file is stored as the "foreign host."

10.1.1. File Transfer Protocols

When a file is moved using an FTP, to make its properties accessible the protocol must transmit them, and the local host's file system must store them. The former is simply a matter of appropriate protocol design. The latter is trivial if the local and foreign file systems are of identical design, manageable if the receiving file system has extensible property support, and impossible otherwise. If the local file system cannot store a property that is associated with the file on the foreign file system, then that property is not accessible to local processes.

Even with extensible property support, intrinsic properties do create problems for file transfers between dissimilar file systems, because two systems may assign different meanings to the same property name. For example, TOPS-20 and Unix have completely different schemes for representing protection information, and each system can represent access rights that the other cannot. This problem will be discussed in more detail in section 11.1. Since there are only a few such intrinsic properties, case-by-case solutions are appropriate.

Even when intrinsic properties have compatible meanings, it might not make sense to give a copy of a file exactly the same property values as the original has. Two reasonable principles are in conflict:

- **FTP transfers should be invertible**: if a file is moved from host A to host B, then back to host A, nothing should appear to have changed on host A.
- **Local file system semantics should be maintained**: the values of intrinsic properties for the copy should be consistent with the state of the local host file system.

Suppose that both file systems maintain a "modification-time" property. Should the copy carry the modification time of the original, or should it carry the time it was created on the receiving system? If the latter, then we violate the first rule, since the property has changed as a result of the transfer. If the former, then the second rule is violated, and the file may, for example, be missed during an incremental backup.

In practice, it may turn out that neither violation is particularly troublesome and thus the conflict can be resolved arbitrarily. We should at least recognize that the conflict exists.

10.1.2. Remote access to files

File access protocols are increasingly supplanting file transfer protocols, partly for efficiency and concurrency, but primarily because they lead to transparency. A FAP should therefore support essentially the same operations as the local interface, so that a program can use either local or remote files. In a heterogeneous system, one FAP must be usable with many different local file systems. A FAP should provide primitive operations that can be composed to yield various interfaces, rather than mimic the interface of a specific file system.

In particular, a FAP must provide read/write access to individual file properties; operations like the Unix *stat* system call can then be composed of several property-read operations. This makes it relatively simple to provide a uniform interface to both local and remote file properties. To support extensible properties, a FAP must bind property names as late as possible: the names must be passed as strings, rather than being taken from some "code book" as was done in the NSW File Package [Cash76].

Access to properties via a FAP runs into the same property name interpretation problems as with FTPs, except that interpretations are imposed by programs, not by a local file system. On the other hand, there is no corresponding invertibility problem, since the file never actually

moves. We believe that remote file access is preferable to file transfer because it avoids some of the procrustean problems of heterogeneous systems.

10.2. Data translation

When files are used across dissimilar hosts in distributed systems, a difference in data formats must often be surmounted. Simply copying the bytes of a text file from an IBM mainframe to DEC minicomputer doesn't make the file usable on the minicomputer; it must be converted from EBCDIC to ASCII, a different end-of-line convention may be needed, etc.

There are several sources of incompatibility. One is different hardware representations for basic data types: word length, floating point format, byte order, and character set. Another is the operating system environment: for example, representation of text files can vary between operating systems even on the same hardware. It may be hard to exchange information both between two hosts with identical hardware but different operating systems, and between two hosts running identical operating systems but on different hardware.

The latter problem has been addressed by LOCUS [Pope81]. Locus is a network of heterogeneous computers all running the same operating system; the processors differ in byte order, data representation, and instruction set. Most files are assumed to be text files and the system automatically converts the byte order when transferring. LOCUS also has special knowledge about directory files, and is able to mask the hardware differences in this case. Translation of executable files is impractical; LOCUS instead maintains parallel versions of programs, and automatically uses the version appropriate to the target machine. This is not a general solution to the heterogeneity problem; it works only because in the most frequent cases, the differences are simple and the system has sufficient semantic knowledge to mask them.

The NSW File Package took a more comprehensive approach. Each file that could be accessed across the network was listed in a master catalog, along with a code that served as an index into a global type table. (Thus, each NSW file had a well-defined type.) The type table had a structural description for each possible type. When a file was copied from one system to another, this structural description was used to transform it at the source host into a canonical format, and then at the destination host to convert the canonical format to one meaningful in the destination environment. Presumably, automatic translation in NSW failed in some cases; for example, there are characters that cannot be unambiguously translated between ASCII and EBCDIC.

NSW could do better than LOCUS because it kept specific structural information about each file. This is an obvious use for file properties. Structural information is distinct from type information; structure is important in representing a file, while type is important in determining how and when to use it.

NSW used file structure descriptions to translate an entire copy of a file. It is better to translate incrementally, so that a file may be shared by simultaneous processes. A file that is updated piecemeal, such as a database, should not have to be entirely translated when only one record is changed. Because of this, and because the file system itself should not be performing translations, translation must be done in a layer between the basic FAP and the database-record access functions in a program.

10.2.1. How often is translation difficult?

We close this discussion by observing that, in practice, complex translation is seldom necessary. Most files are either text files, files with simple, portable structures (such as font files), or files for which translation is meaningless (such as executable programs). Large databases may account for a lot of disk space, but they are seldom accessed directly as files.

Table 10-1 shows the distribution of file types found on a typical Unix system. The initial classification into types was done using the *Unix file* command, which uses various heuristics to guess the type of a file. These initial classifications were then refined by hand, mostly using knowledge about file naming conventions; only 3% of the files could not be classified.

File type	Number of files	Percentage of all files		
English text*	5965	8.36%		
Troff-formatter input text*	2948	4.13%		
Shell scripts*	2631	3.69%		
Program sources*	5926	8.30%		
Unclassified text*	37223	52.15%		
Subtotal: text files			54693	76.62%
MC68000 programs/objects	554	0.78%		
Xerox Alto boot files	81	0.11%		
Xerox Dolphin microcode	4	0.01%		
PDP-11 LDA format	15	0.02%		
Compiled Lisp	45	0.06%		
VAX stand-alone	63	0.09%		
Subtotal: foreign program binaries			762	1.07%
VAX Object libraries*	124	0.17%		
VAX Object files*	3331	4.67%		
VAX Executables*	1754	2.46%		
Subtotal: native program binaries			5209	7.30%
Core dumps	41	0.06%		
Subtotal: program binaries			6012	8.42%
C/A/T format*	32	0.04%		
Xerox Press format*	446	0.62%		
DVI format	826	1.16%		
Subtotal: printer intermediate formats			1304	1.83%
Fonts	2266	3.17%		
Xerox SIL format pictures	37	0.05%		
Xerox Draw format pictures	110	0.15%		
Subtotal: printing formats			3717	5.21%
Devices*	293	0.41%		
Directories*	3135	4.39%		
Symbolic links*	562	0.79%		
Subtotal: non-file entries			3990	5.59%
Empty files*	655	0.92%		
System accounting data	156	0.22%		
Compressed text	5	0.01%		
Encrypted passwords	448	0.63%		
VMS-format files (>50% text files)	89	0.12%		
Miscellaneous/unclassifiable	1638	2.29%		
Subtotal: others			2991	4.19%
	Total		71383	100.00%

* Classified automatically by *file* program

Table 10-1: Distribution of file types on a typical Unix system

More than 76% of the files in this system are text files, easily moved between systems once it is known that they are text files. Another 8% are executable programs, which may be copied without translation since they are only meaningful on the host for which they were compiled[2]. About 5% of the files are printer intermediate format files; these formats have been designed for portability, making translation unnecessary.

Aside from these three categories, and those directory entries that do not actually refer to files, less than 4% of the files might actually require non-trivial translation when transferred between hosts in a heterogeneous system. The huge majority of files can be transferred either verbatim, or with simple translation, such as converting end-of-line sequences in text files; although translation in general is difficult, the specific cases for which it is easy are prevalent by far.

When we know the types of files, we know when we can apply one of these simple transformations. If type and structure properties are available, almost all files can be translated automatically if they can be translated at all. If these properties are not available, automatic translation turns into a hard problem.

10.2.2. Property translation

File systems within a heterogeneous distributed system will have certain differences that simply cannot be resolved. Most important of these are differences in protection models and the corresponding protection properties. Some of this is due to differences in the assignment of function to the file system; execute-only (unreadable) files and Unix "setuid" [Ritc78] files are difficult even to conceptualize in a distributed system where programs may execute in administrative domains disjoint from the file systems where they are stored. It is also due to the inherent incommensurability of certain protection models, such as access control lists and capabilities.

In a heterogeneous distributed system, we must accept these mismatches, and concentrate on solving those problems that can be solved. Some properties, such as "length in bytes" or "modification time," can be translated between certain pairs of systems. With file transfer protocols and file migration, the best we can do for untranslatable properties is to store them in an accessible form; for this, we need extensible property support. If we use a remote file access protocol or are porting a program between file systems, we need only provide uniform access to properties; translations can be done, if necessary, by specific applications.

10.3. Migration

Many distributed systems go through a lot of effort to make the location of files unimportant to the user; this is the transparency paradigm. Sometimes, though, the location of a file is important to the user, and a good distributed system makes it easy to put the file in the right place.

Moving a file from one place to another to achieve some policy objective is called *migration.* Migration can be used to:

- Place a file closer to the process that is using it, to avoid the cost of repeated access over the network.

[2]Note that a significant portion of the executables stored on this VAX system are for foreign (non-VAX) processors; this demonstrates how convenient it is to transfer foreign executable files.

- Archive a file to cheaper storage, in the expectation that it will not be used again.
- Restore a file from archival storage when it is needed again.
- Save a file on more reliable storage.

Some of these goals can be obtained by using a transparently replicated file system, but replicated systems are not yet universal.

Migration requires answers to these questions:

1. How is the file moved without damage? What protocol is used to move the bits, and how are they stored once they arrive?
2. When and to where should a file be moved? Ideally, an oracle would know in advance where the file should be. Lacking an oracle, we must base migration decisions on easily predictable access patterns, resource allocation policies, and user direction.
3. How can we maintain transparency? We would like to maintain the transparency illusion for those processes and users not concerned directly with the migration policy. After a file is migrated users should be able to access it as before, and access should not be interrupted even while the file is being moved.

The third problem can only be solved if we can transparently migrate processes in a distributed system, and is beyond the scope of this thesis[3]. We can address the other two questions, so far as they concern file properties.

10.3.1. How to migrate a file intact

When a file is migrated, its properties must be migrated with it. As far as possible, migration should be invertible without affecting any of the properties (we saw in section 10.1.1 that this is not always practical). Thus, the file system at the site to which the file is migrated must be able to store all of die properties associated with the file; invertible migration in a heterogeneous system is impossible unless the target file server supports extensible properties.

A file being migrated to archival storage should not be translated. It will probably not be accessed directly on the archival server, but will be moved back to the original system before being used again. Since some files cannot be translated automatically, requiring translation would make it difficult to archive them.

If a file is being migrated to put it closer to the using process, it is unlikely to be an untranslatable file such as an executable program (unless the two processors involved are identical and translation is not required). It should probably be translated during the transfer, to avoid much higher translation overhead at run time.

During migration, incompatible intrinsic property values may have to be translated. There are really two kinds of file migration: migration between active systems, where translation might be required, and migration between levels of storage, where translation might be impossible. If a file is being migrated to archival storage, where it will not be accessed directly, we can "encapsulate" all intrinsic properties as non-intrinsic properties before migrating them; they can then be restored when the file is retrieved. If a file is being migrated between active systems, then the incompatibility problem must be faced.

[3] See Powell and Miller [Powe83] orTheimer [Thei86].

10.3.2. Making the migration decision

The choice of when and where to migrate a file is a policy decision. File properties can provide information on which to base this decision; they can store information on access patterns necessary to make automatic decisions, and information on user intentions necessary to avoid making the wrong decision.

We know from research on virtual memory page replacement algorithms that it is possible to estimate from its history the likelihood that a page will be needed in the near future; the working set model is an example of a useful estimator [Denn68]. The policies that work for pages might not work for files, but they suggest some approaches. For example, when deciding whether to migrate a file to long-term storage, some file systems move files that have not been accessed recently. One might also base a migration decision on how frequently a file is used, how large a file is, etc. These policies and others have been analyzed for a number of real or synthetic file-access distributions, and for a variety of different optimality measures [Stri77, Smit78].

Automatic migration, by whatever policy, requires some information about the reference patterns for each file. LRU needs only the time of last access, already maintained by most file systems; Unix, however, does not always maintain this consistently [Free84]. More detailed reference information could be kept in other properties, at negligible added cost.

While frequency-of-use and time since last access are reasonable estimators for "average" files, there are situations in which they lead to the wrong decision. It is worse to err on the side of unwanted migration: some infrequently-used files are needed in a hurry when they are needed at all (the emergency procedures for a nuclear power plant, for example). A client should be able, using properties, to mark some files exempt from migration.

Users often know in advance when they will next need a file, and could advise the file system to have it ready at the appropriate time. Some examples: a payroll database might be needed on-line once every Friday, but otherwise could be kept on cheaper storage; students' records might be archived one year after they graduate; a songbook of Christmas Carols might be archived in January with instructions to restore it in November. A "when-needed" property would not only ensure that a file is available when it is needed, it would also make it possible to migrate it immediately when it is unneeded.

Chapter 11
Future Work

In this chapter, we look at some unresolved problems with file properties. We start, in section 11.1, with the problem of naming properties and associating semantics with property names. The next two sections cover some details of performance and functionality: how to find string-valued properties by pattern matching, and improved support for query optimization.

Section 11.4 discusses the practical problems of adopting property-based file systems, and suggests some approaches to integrating them with existing systems. Finally, section 11.5 looks at a class of applications, all involving relationships between files, and shows how they might make use of property-based file systems.

11.1. Naming and Semantics

The property support facilities proposed in chapter 6 are mostly syntactic: except for intrinsic properties, the file system knows nothing about the property names and values that it stores. The semantics of all user-defined properties must be defined at a level above the file system.

Given a conceptual attribute of a file, how do we express it as a file property? How do we choose a name for the property? And, given a file property, how do we understand what it means?

These are not easy issues to resolve, even in the relatively simple "language" of file properties. In this section, we try to expose the problems and present partial solutions. More complete solutions must await more extensive experience with full file property support.

11.1.1. The relationship between meaning and name

The central problem of file property semantics is the connection between meanings and property names. Since we cannot anticipate the actual meanings of potential properties, all we can do is to try to establish a consistent and convenient scheme for assigning property names and deciding what agent should interpret them. We must avoid both ambiguity and uncertainty.

11.1.1.1. Making use of human intelligence

It is not an accident that properties, in our system, are identified by human-sensible name rather than by some more "efficient" scheme. When a program is incapable of interpreting a particular property, a human user may still be able to deduce its meaning. After all, the property name was chosen either by another human, or by some scheme invented by a human programmer.

Property names should always be constructed so that their meaning will be clear. There is no reason to use a name such as "IEH138" when a name such as "Error-Code-of-Creating-Program" could be chosen instead. The PLDIR implementation is designed to encourage the use of long, descriptive property names, by not imposing a performance penalty for their use.

11.1.2. Name management problems

We identify three kinds of property-name management problems: the use of multiple names for the same concept, the assignment of multiple meanings to a single name, and the possibilities for divergence created by the use of heterogeneous distributed systems.

11.1.2.1. Multiple names for one meaning

In natural language, the use of synonyms causes no problems if the vocabulary of the audience is sufficient. Synonyms do cause problems in non-computer filing systems, card catalogs, and indices; if we desire a recipe for "garlic mayonnaise" but do not know French, we might not find it indexed as "aïoli."

Similarly, if a program wants to know the "length-in-bytes" of a file and the file instead has only a "size-in-bytes" property, we cannot expect it to recognize the synonym. It is important for applications that share, or that could potentially communicate through, a property to have a common name for it.

This is not a severe problem for intrinsic properties, since file system designers publicize these names, nor is it a problem for a closed set of applications designed to communicate with each other. It is most apparent when two independently-designed applications maintain or use similar information.

11.1.2.2. Multiple meanings for one name

A single name should not denote multiple meanings. Homonyms in natural language cause confusion if they cannot be disambiguated using context. Similarly, if a program believes that the "size" property of a file denotes its length in bytes, and instead the property denotes the file's size in disk blocks, serious errors may occur.

Homonyms are more dangerous than synonyms. If two applications use different names for the same concept, the worst that can happen is that they fail to communicate. If two applications use the same property for two different concepts, one might assign it a value that the other, interpreting it in a different way, could use to make an incorrect decision.

11.1.2.3. Heterogeneous distributed systems

In a heterogeneous distributed system, the management of names is dispersed among authorities separated by space, time, and institutional culture. When mutually oblivious naming schemes are joined into a heterogeneous distributed system, both homonyms and synonyms may appear.

Imagine two distributed systems with no connection between them. Each has a naming authority that has been managing the assignment of names for user-defined properties. Now suppose the two systems are joined together: the two naming authorities may have assigned property names that now conflict. This can happen even if the systems use homogeneous file system software.

It is also possible that the joining of previously unlinked systems into a heterogeneous network can create conflicts between intrinsic property names, or between intrinsic names in one system and user-defined names in another.

11.1.3. Name management solutions

In this section, we examine several name management systems. They vary in the amount of rigidity they impose and in the likelihood that they will allow conflicts.

- **Anarchy**: The simplest scheme; there is no need to communicate with any authority before using a new property name, and so there is no delay associated with property name allocation. Anarchy is sure to bring name conflicts, and provides no mechanism for resolving them.

- **Centralized name allocation**: This can almost guarantee unambiguous and unduplicated name assignments. It cannot be perfect: we are unlikely to have formal specifications for property meanings, so misunderstanding is always possible. Central management is the only solution that can avoid synonyms; all the other solutions can at best avoid homonyms.

  The need to communicate with a central management might make it extremely difficult to acquire a new property name, and probably would involve a long response time. This might stifle clean solutions to "small" problems, just as non-extensible file systems force programmers to misuse existing properties.

- **Distributed name registration**: A distributed database system that allows any user to register a property name and a brief description of its meaning would prevent two users from choosing the same name for grossly different purposes. Mechanisms, such as keyword lookup, to search for concepts that have already been assigned names would help prevent the use of synonyms.

  With distributed name registration, either there must be a restrictive protection scheme, which suffers from some of the flaws of centralized management, or it is possible that a malicious, inconsiderate, or tasteless user could register a property name with an inappropriate meaning. Also, distributed registration does not prevent conflicts when two systems are joined into one.

- **Naming conventions**: A hybrid of the centralized and anarchy schemes is the use of officially promulgated, but unenforced, conventions for naming properties. For example, keyword properties would conventionally begin with the prefix "Keyword-". Application-specific property names might include the application name; for example, "USENET-expiration-date". With a small set of similar conventions, recourse to a central authority could be reserved for the situations in which it is actually needed.

  Conventions also aid human users in guessing the meaning of unknown properties.

11.1.3.1. Structured names

We can build on the idea of naming conventions by establishing a formal structure on names, similar to the hierarchical structures used in the Clearinghouse [Oppe83] or in the Internet Domain Naming scheme [Su82]. These schemes impose a hierarchical classification structure on the space of possible names; at each level of the hierarchy, names may be freely assigned without involving higher or neighboring levels.

The highest level could be divided into classes such as "Intrinsic," "System-Utility," and "Private," and each of the next levels divided according to function, application, or organization. For example, one name might be "Intrinsic.Unix.Protection"; another might be "System-Utility.USENET.Expiration-Date"; a third could be "Private.JoeSmith.PhaseOfMoon".

At each level in the hierarchy, the view upward is a centrally-managed one, while the view downward is left to the discretion of the lower-level authorities. This system avoids multiple uses for a single name, without unduly restricting the assignment of names for small applications. It also indicates which sub-authority assigned a name, in case the meaning is not obvious.

The apparent drawback of this scheme is the length of the names it generates[1]. Some of this can be reduced by adopting cryptic, but widely understood, contractions for high-level names; e.g., using "I" for "Intrinsic." It may also be possible to automatically disambiguate abbreviations at the user-interface level: "Protection" may be taken, on a Unix system for example, to mean "Intrinsic.Unix.Protection". Display-oriented user interfaces might eliminate most occasions on which users would have to type long names.

A carefully crafted scheme of structured property names is probably the best compromise between rigidity and anarchy.

11.2. Pattern-based retrieval of string properties

Clients may sometimes perform queries on string-valued databases using predicates more complicated than an equality test. For example, one might ask for those values lying alphabetically between "Jones" and "Smith", or those values matching a pattern such as "b*cd?"[2]. One could imagine predicates that are context-free grammars (or even context-sensitive).

The choice of whether to support such predicates is a tradeoff between their expected utility and the cost of implementing them. Because we expect directories to be small (and file histories to involve a relatively small number of versions), queries over a single directory or file history can be done using linear search, even with complex predicates. The database organization of our PLDIR implementation does not provide an efficient basis for evaluating complex string queries over an entire file system; this would require different data structures.

One simple form of pattern is the conjunction of substring queries. If the individual substring queries are sufficiently restrictive, then it is reasonable to execute them and merge the results. The substring queries might be implemented using *position trees* [Aho74], which allow fast insertions and lookups for substrings using space linear in the amount of string information stored; deletion may prove expensive, though.

If the individual substring queries are not sufficiently restrictive, then this approach will not work well. For example, although the pattern "e*e*e*e" may match only a single value, it may not help much to find those values containing the substring "e". The feasibility of even such limited pattern-matching thus depends on assumptions about the set of values and the likely queries.

It is somewhat easier to implement predicates that specify a subrange of a lexically-ordered

---

[1]Recall that the PLDIR implementation does not impose a performance penalty for long property names.

[2]This follows the pattern conventions of the Unix shell, in which '?' matches exactly one character and '*' matches any substring, including the empty one. Such patterns are less general than regular expressions, because they don't allow for repetition of substrings.

set of values. Position trees might be suitable, thus supporting subrange and pattern queries with a single database. Alternatively, the *trie* [Aho83] data structure requires linear storage space, performs insertions and deletions in time proportional to the length of the string, and (as does a position tree) groups lexically adjacent entries, allowing for efficient retrieval of subranges.

That it is feasible to support string subrange, substring, or pattern queries does not mean that they should be provided. In fact, we cannot provide examples of file properties for which such queries are necessary; problems that at first appear to require them can usually be reformulated to use data types other than character strings. Experience with property-based file systems may uncover instances of appropriate applications, but it does not make sense to support complex queries until the need for them is demonstrated.

11.3. Better support for query optimization

The cost of evaluating conjunctive queries can depend strongly on the order in which the simple queries are performed. For example, take the query * 'what text files were last modified between 1:00 and 2:00 yesterday?" One could evaluate it by first obtaining a list of text files, then checking the modification time of each. As we showed in table 10-1, more than 75 per cent of the files in a typical file system might be text files, so this evaluation order is wasteful. It would be better to obtain a list of files modified during the specified period, and then select the text files from this list, since we expect that relatively few files were last modified during the specified period[3].

Choice of optimal query evaluation order, in the general case, requires relatively deep understanding of the semantics of the databases. A human user, with implicit knowledge of the databases, might well be able to make the choice described above, but if we want programs to optimize evaluation order, semantic information must be explicitly available to them. When this requires estimation of the relative sizes of the range and domain of a query, such as "most files are text files,[1]' we enter the realm of artificial intelligence or expert-system approaches.

Certain properties, however, are possessed by a relatively small number of files. For example, if the query instead was "what text files have the 'Number-of-Eggs-required' property equal to 3," a program would not need any semantic understanding to choose an evaluation order. Instead, it would simply notice that only 17 files have the "Number-of-Eggs-Required" property at all. As a rule, subqueries with a very small domain should be performed first.

We can therefore support certain effective query optimizations at very low cost, by providing a *GetIndexCardinalityQ* function as part of our server interface. This function would return the number of entries in a specified index; it is easy to implement this efficiently. Even an "intelligent," semantics-based query optimization system would be able to make good use of such a function [Smit85].

[3]On a typical file system containing user's personal files, less than 1% had modification times between 24 and 48 hours old; the fraction last modified during a single hour of that period would be even smaller.

11.4. Adoption of property-based file systems

We argued in chapter 10 that extensible property support is necessary for file transfer and migration in heterogeneous distributed systems, and that a uniform property-access mechanism aids remote file access and program portability. Ideally, every member file system in a heterogeneous distributed system would have uniform, extensible property support, but this is an unrealistic hope.

We can nonetheless derive benefit from integrating property-based systems into existing heterogeneous systems. Some of this benefit comes from the asymmetry of file migration; one can migrate files from a traditional file system to an extensible one without any special property support in the traditional system. Some comes from the relative ease of introducing uniform access, allowing the development of portable and remote-file-access applications. Finally, we would hope that the superiority of the extensible-property file system would be obvious enough to encourage its adoption throughout the system.

Adoption of the property-based approach into a real heterogeneous system would have to be done incrementally. Introduction of property-based file systems might start with the use of compatibility packages, such as the UFE front-end for Unix (described in section 7.4), to provide uniform access to properties within a heterogeneous system. Simultaneously, we should introduce a remote-file-access protocol (FAP) that uses this uniform mechanism. Neither of these steps is particularly difficult, nor do they affect system performance, yet they enable program portability and heterogeneous remote access to properties.

The next step is to produce programs to take advantage of uniform access. New programs should be written to use the uniform file system interface. Old programs could be modified to do so, but in many cases it would be sufficient to re-link them against an I/O library that simulates the old file system interface using the uniform interface. Such a compatibility package would be even easier to implement than UFE.

Extensible property file systems could then be introduced into the heterogeneous system. At some point, however, one would have to face the incompatibility of applications that take advantage of extensibility with the remaining non-extensible file systems. The most direct solution would be to replace the non-extensible file systems with extensible ones; this is not going to be easy. More likely, the sophisticated applications will gravitate towards the extensible file systems, leaving the older ones to support existing applications.

11.5. File Properties and Relationships Between Files

In the world of paper files and documents, few works exist in isolation: scholarly papers cite references, legal decisions cite precedents, invoices refer to purchase orders, owners' manuals come with new cars. Many computer files also do not exist in isolation, but are abstractly connected to one another: on-line help texts are associated with specific programs, sources files with object files, memos with replies.

The concept of a property list arose in Lisp, where it was needed to store information about atoms not derivable from their relationship. We can draw the complementary lesson from Lisp: structure is information. The relationship between files can be vitally important in the use of their contents; a file system should make it possible to record these relationships.

How is this done in the paper world? Typically, for a document of a certain type there are conventions that allow one to find the references to related documents. There are well-known rules for citations in scholarly papers; invoices have fields marked "order number" and

"customer purchase order number"; owners of new cars know to look in the glove compartment to find the manual. Unfortunately, the rules are different for each kind of object. This kind of *ad hoc* mechanism has been used in the computer world for lack of anything better; if an application requires a reference between files, an application-specific mechanism is used. Moreover, it is often difficult or impossible to invert these references; that is, given a file, to find the other files that refer to it.

This is unsatisfactory. Just as a computer file is more than an abstract disk, it is more than an abstract piece of paper and a computer file system is more than an abstract file cabinet. Why should we simulate primitive paper-world techniques to record and manipulate relationships when we have the power of a computer system available?

We can use file properties to record relationships uniformly and cleanly; we can use the query facilities of our file system design to efficiently invert the relationships that have been recorded. In section 11.5.1, we show how to use the designs of chapter 6 to record, discover, and invert relationships. In sections 11.5.2 through 11.5.5, we look at the application of relationship knowledge to a variety of problems. Finally, in section 11.5.6, we look at the relationships between files and smaller pieces of unstructured information.

11.5.1. How to represent relationships

How can we use the file system design of chapter 6 to represent relationships between files? There are four pertinent issues: how to name the referent file, what kinds of relationships can be represented, how to represent relationships of each kind, and how to invert the references.

11.5.1.1. Naming **the referent file**

We distinguish between *directory references* to files, which are references held by the directory system, and *secondary references,* which are references held elsewhere, such as in program sources, file contents, file properties, or scraps of paper. Properties denoting relationships between files are secondary references.

A name appearing in a property as a secondary reference to another file must be interpretable with respect to a global name space. Relative names do not work because there is not necessarily a context within which such a name could be interpreted. This is especially important in a distributed system, since inter-file references may cross file system boundaries.

There are two kinds of global names for files: low-level unique identifiers (UIDs) and complete path names. (Sometimes, these are called "hard links" and "symbolic links," respectively.) The difference between UIDs and path names is one of binding time. UIDs are bound early; a hard link refers to a specific instance of a file. Path names are bound late; by indirection through the directory system, a symbolic link refers to the file version currently associated with a name, and the version may change. One uses a hard link when referring to a file with specific contents, a symbolic link when referring to a file with a specific purpose. A hard link is used when the reference should be to a specific file version even if it is superseded, while a symbolic link is used when the reference is to the "latest" version of a file.

Path names are superior to UIDs for use in secondary references to files; UIDs should only appear within directory entries, because:

- The binding-time effect of hard links can be simulated with symbolic links, using appropriate conventions; the converse is much harder.
- The directory system will map a path name onto a UID; it may be quite difficult to map a UID onto a path name.

• Using UIDs in file properties unnecessarily exposes low-level details to humans.

UIDs are not necessary in secondary references; path names are. File properties that refer to other files should denote them by path names, relatively to a universally-known root.

11.5.1.2. Kinds of relationships

Relationships between files can be one-to-one, one-to-many, many-to-one, or many-to-many. Whether a relationship is one-to-many or many-to-one is a matter of point of view; for example, if many programs depend on a single library file, we could also view it as a single library file being referenced by many programs.

A relationship has a direction, since we cast relationships in terms of references. That file *A* refers to file *B* does not necessarily imply that file *B* refers to file *A*. Since we can create inverted indices on the value of any property, the direction of a relationship matters only in that it may influence the relative cost of representing it one way or another. In many cases, only one direction is interesting, and we can dispense with the inverted index.

Table 11-1 shows a few examples of relationships between files. The relationship between a program and its "help text" is one-to-one; between documents in a structured text system is many-to-many; between chapters and a book is many-to-one or one-to-many, depending on the point of view.

From	To	Comments
program	help-text	
object file	source file	To know when to recompile
program	object file	To know what to recompile
memo	reply	
software release	components	Avoid omitting files from release
chapter.mss	whole.mss	Scribe "@Part" command
whole.mss	whole.aux	Scribe "auxiliary" file
program	kernel	Ensure program matches kernel version
program	configuration file	E.g., /bin/csh reads ~/.login
core dump	program	
gmon.out	program	
program	makefile	
makefile	source files	
database format	programs	
document	document	Structured text
documentation-node	documentation-nodes	Documentation browser

Table 11-1: Examples of relationships between files

11.5.1.3. Representing the relationship

How do we represent the four kinds of relationships? Starting with a way to represent many-to-one relationships, we can represent all four classes. Take, for example, the relationship between a set of programs and an object library against which they are linked; we want to know whether the programs must be relinked because the library has been updated.

We store the information on the property lists of the programs, instead of property list of the library. The latter would be inconvenient because any user who wishes to "register" a new program would need access to the property list of the library. Further, we are reluctant to use of multi-valued properties (such as a string encoding a list of dependent programs), since they are costly to store and update.

The property cannot simply be called "depends-on", since a program might depend on several libraries; again, we do not want to use a multi-valued property. The solution is to create

a property name based on the "target" file name, following a rigid convention so that all files with such a property will use identical names. For example, a Unix program would have a property named "depends-on-/lib/libc.a". The property value is merely a placeholder.

Suppose we want to reverse the point of view, making this a one-to-many relationship, so that whenever we update the library we can determine which programs must be relinked. We need only create an inverted index on the given property name; we need not create additional properties.

One-to-one relationships are represented as the obvious special case of many-to-one relationships. Many-to-many relationships also are easily represented using the many-to-one scheme, since we chose not to use a single "depends-on" property name.

11.5.2. Non-linear text

If you study the
logistics
and heuristics
of the mystics
you will find
that their minds
rarely move in a line
— Brian Eno

Traditional computer files are good at representing single texts, such as memos, articles, and books. They are not that good at representing structures of related texts, such as a multi-party discussion on a computer bulletin board, the trail of drafts and revisions leading up to a published book, or the users' manual for a complicated system.

The concept of "non-linear" text has been introduced as a reaction to this problem. The central idea is that a document, or other coherent whole, is made up of a network of small pieces of "linear" text. In chapter 3, we mentioned "Hypertext" [Nels65], NLS [Enge68], PIE [Gold80, Gold81], and Notecards [Xero85a]. Another related current, much better developed, is that of structure-based or "syntax-directed" program editors [Teit81]. These reject the traditional notion of a source-language program as a string to be parsed after editing is finished; instead, the editor operates directly on the parse-tree, and the textual representation is displayed only for the user's benefit.

Non-linear text has proved useful in the construction of sophisticated "help" services. A user navigates from node to node in a network of information; each node typically describes one feature or function of the system being used, and contains cross-references to related features and concepts. Instead of having to continually refer to an index and a large set of paper manuals, the user of a carefully-constructed network can quickly find the relevant information. An example is ZOG [Robe79]. One goal of PIE is to provide structured help integrated with program sources.

A significant problem in this area has been granularity: how large should the linear pieces be? Many structure-based program editors allow users to enter expressions in linear form, as a concession to human psychology; others treat terminal symbols as the granules [Meyr82]. In the Unix *man* system, the documentation for one command is the basic unit, while NLS, Hypertext, and PIE treat objects at about the paragraph level as atomic. Notecards is intended

to treat "ideas" as atomic; an idea may be as brief as a bibliography entry or a footnote. A paragraph is probably the smallest granule that may practically be placed in a distinct file; one could construct a paragraph-level system as a layer above a file system, using properties to represent links between the objects. The PIE and Notecards systems, as part of the SmallTalk [Gold83] environment under which they run, both make use of the SmallTalk "object" concept instead of a file system.

Text of sub-paragraph granularity leads us into the "small-object" problem: can we represent arbitrarily small permanent objects in a file system without too much overhead, or is this better solved using a database? The problem is beyond the scope of this thesis, although the discussion of annotation in section 11.5.6 touches on a special case.

11.5.3. Auxiliary files

One way to improve performance is to avoid recomputing results: optimizing compilers use common-subexpression elimination, and processor designers use cache memories. Many applications that involve processing a data file repeatedly can use a similar strategy. We call files that store precomputed information about other files *auxiliary files*.

For example, in a modular programming language it is not necessary to recompile an entire program when a small part is modified; only the module changed must be recompiled. The inter-module links must be resolved, but this is not nearly as expensive as compilation. Precompiled object modules are thus caches of a sort; although the ultimate truth resides in the source code, the cache short-circuits most of the recompilation process.

A different use of an auxiliary file is to store index or "table of contents" information about a file. This is not used to cache the result of a complete pass over the file; it is used to avoid a complete pass when only general information is needed. For example, a table of contents for an object library eliminates the need to search the entire library to find a symbol. The Scribe [Reid80] document compiler stores an outline of the document being processed, so that on subsequent runs forward references may be resolved in a single-pass algorithm, if changes to the source have been minimal.

To use an auxiliary file, one must be able to find it starting from the ultimate source file. In current systems, this is done by giving the auxiliary file a name similar to the source file, an instance of name-encoded property information. A more robust method of maintaining the association between source and auxiliary file is to store this kind of relationship in file properties; this would work even if the source file is renamed. Also, since auxiliary files are of direct interest not to the user but to the tools that use them, they should not clutter up the user's name space (directory). By using properties instead of names to link the files together, the auxiliary files could be hidden in a separate context.

11.5.4. Compilation management

Most interesting computer programs are made up of many pieces, often written by many programmers. Putting these pieces together isn't easy, especially if they can be put together in different ways to produce difference versions of a program. Management of the compilation process requires sufficient information about the components; since these are usually stored as files, file properties can be put to good use describing relationships between the components.

The Unix *make* program [Feld79b] takes a simple approach. *Make* reads a description file to determine what components are needed to assemble a program and how they depend on one

allotne r; the dependency information is combined with information on file modification times to fST^^ must be recompiled. ^^ almoStno Semandc demanding of X^£ £- it deals with, and so can be used with programs in practically any language but this ignorance means that *make* must be conservative when deciding what needs i 4 2 & ^ l nd that the dependency information must be provided manually.

. ^ - S ^ t e " Moddli^ hi^uage (SML) [Schm82, Lamp83J, used with the Xerox Cedar system, takes a distinctly different approach. SML is an applicative subset of Cedar used to describe how to compose a set of related system configurations from their elements." Since SML is intimately related to the programming language, SML-based tools can decide with arbitrarily high precision what needs to be recompiled. The SML approach can only be used with a single programming language, however, and cannot be applied to many popular Ianguages. J r r

The "smart recompilation" method described by Tichy [Tich85] is similar to SML in that it depends on intimate semantic understanding of the source code; it generates records of the form "module X refers to identifier A defined in module Y." These are attached to the referenced modules, so that when a module is changed, it is easy to find the modules depending on it. (There might be a lot of these records, but they can only increase object module storage costs by a small constant factor.) A user who creates a new "referencing" module must therefore update the reference set of the exporting module Y; this complicates access control. Because of the global query support in our file system design, we could instead record the "refers to A in Y" records as properties of the referencing module X, and "smart recompilation" could be implemented without bizarre access controls.

Multi-language programming environments need something between the simplistic approach of *make* and the highly integrated SML approach. *Make* and SML share the concept of a configuration description, although for *make* the granules are always files, while for SML the granules are programming language constructs. A possible synthesis of the two approaches would be:

- **Describe module structure in a description file:** The description file would still describe the structure of the modules that constitute a program, but would refer only to source modules and libraries, not to intermediate code.
- **Remove dependency information from the description file:** A compiler knows when it generates an object module how the object module depends on source files. These dependencies can be recorded as properties of the object module.
- **Record "export lists" in file properties:** A structure editor or preprocessor could record the names of exportable objects in the source file's property list. (The compilation management system would invoke the preprocessor when it notices that the source file has been modified.)

Each importation of a symbol would be represented by a property named "Imports-<symbol>", whose value is the name of the source file from which the definition was obtained. Each exportation of a symbol would be represented by a property named "Exports-<symbol>", whose value would be the time the symbol's definition was last changed. When a structure editor or version-control system is used, these timestamps could be exact; otherwise, they would reflect the most conservative estimate.

The compilation management system itself can thus function without any language-specific understanding. The language-specific code is left in the compilers and structure editors, where it belongs.

If a collection of software modules is treated as a closed system, then the import and export information could be stored in a centralized database [Lint83J. But most large software projects

are not closed systems; pieces are exchanged with other organizations. With the export information attached to each source module in its property list, it would be easier to preserve such information when exchanging files.

11.5.5. Release Support

One problem in software product management is to make sure that "releases" contain all the necessary components. When a software system is extracted from its development environment, unnecessary files should be left behind, but it is often difficult to know exactly which files are needed. A common technique is to create a release as carefully as possible, and then install it on a test site to see if anything has been omitted. As with any test-based verification technique, it is hard to be sure that all paths through the system have been exercised, and missing files are often found once the system has been shipped. Also, this method depends on the availability of a separate environment to be used for release testing, a potentially expensive requirement.

Release of systems including source code can be partly verified by attempting to recompile every program. One must still verify that the recompilation procedure has in fact recompiled all the programs in the system. This approach is not sufficient for releases that include non-program files; it cannot verify that all files have been included in a release. For example, a typesetting program might depend on a library of font information.

We need a formal description of the dependency relationships between all files in a release, including non-program files. One could then start with a list of the major components of the system and compute the transitive closure of these relationships to discover the files to be included in the release. Some manual effort would be required to set up the links between files, but these would be local to each component of the system, and easier to manage and maintain than a global list of files to be included.

The "description files" (DF) system [Schm82] can describe releases that include non-program files. DF can be viewed as a partially centralized scheme for representing inter-file relationships; we could use our file-property "database" as the basis of a DF-like system.

A similar problem can be solved the same way: when a software system is installed, there often are site-specific configuration files that must also be created and installed. For example, a mail-handling system might have a configuration file to control forwarding of messages. People who install software systems often forget to install these files; system-installers do not always understand what they are doing. An automated procedure, to check that site-specific files exist and are up-to-date, would avoid a lot of confusion.

11.5.6. Annotation

When people use paper files and documents, they often mark them with short notes. The increasing popularity of such products as Post-It[4], pads of small slips of paper with gummed backs, demonstrates how useful this is. A note might be made on a memo of a relevant phone number; a telephone bill might be marked "pay by October 3"; a report might be marked "make 10 copies."

---

[4]Post-It is a registered trademark of 3M.

These notes are meant to be seen immediately when someone handles the document, yet not treated as part of the document. We should be able to attach similar "annotations" to computer files, to be treated in the same way. It would be nice if a text editor, file viewer, or browser, when reading an on-line document, displayed an attached annotation, such as "this document isn't quite finished"[5].

These notes, either in the paper world or the computer world, are typically brief — about one line of text — and could be stored in a single string-valued property. More extensive notes should be attached by using the property to name a text file. Browsers would recognize this escape mechanism and find the associated file automatically.

Why not simply insert annotations into the file itself? We already do this when we place comments in source code or in drafts of documents, but in general inserted annotations suffer from the limitations of embedded properties, as discussed in section 5.1: for example, one might forget to remove annotations from a document before printing it, or one might be annotating a text file that must be in a specific format because it will be read by a program. The use of properties for annotation makes it possible to evade the rigid structures to which many files must conform.

[5]One commercial product called *Note-It* [Reed86] already does this in a rather limited way, allowing users of the *Lotus 1-2-3* spreadsheet to attach notes to spreadsheet cells or file names. Note-It can search for notes containing a specified word or phrase, in effect providing property-based location of spreadsheet cells.

Chapter 12
Conclusions

We divide the contributions of this thesis into three categories: what we have discovered, what we have constructed, and what we have proved.

12.1. What we have discovered

The primary argument of this thesis is that a file is not a homogeneous piece of data standing as a virtualization of a disk or tape, but an object with properties that are central to its being. These properties must be explicitly available for manipulation if programs are to deal intelligently with files.

The evidence for this position is our observation that many seemingly unrelated problems in computer systems are in fact manifestations of the lack of adequate support for file properties. We have seen this in a number of ways:

- Existing systems are unnecessarily non-uniform; this prevents program portability and heterogeneous sharing.
- Most existing systems are not extensible; this requires programmers to rely on *ad hoc*, inelegant, and usually inadequate mechanisms to represent file property information that was not anticipated by the file system designer.
- Existing systems do not provide efficient property-based search facilities; hierarchical directory systems are insufficient to locate files that might be classified according to non-hierarchical criteria. This precludes many applications and makes others distressingly expensive.

We have suggested how a diverse set of problems might be more efficiently or conveniently solved by taking advantage of carefully designed file property support.

12.2. What we have constructed

To put our ideas in concrete terms, we constructed a model of file properties, specific designs for real directory and file systems, several implementations of the interesting parts of these designs, and applications of the designs to specific problems. Each of these constructions is meant to illustrate the application of the file-property paradigm, not to constrain its interpretation.

Our model provides a framework around which systems supporting file properties can be designed. The value of this framework is that it promotes compatibility among a space of possible designs; all designs in this space will include at their cores a single set of primitives.

Even if the details of the designs vary, it will still be possible to connect them in distributed systems and share files, or to write programs that can be ported easily from one system to another.

Our directory and file system designs show how file-property support can be integrated into a real system, and can give a programmer a feeling for how it might be used. We believe that these designs could be used for a production-quality file system, or could be adapted through slight modification for use with similar systems that do not now support properties.

Our implementations of the directory system design indicate how such a system can achieve reasonable performance. The PLDIR implementation, using a special-purpose database, was useful in quantifying the performance of property-based systems, and in demonstrating the advantages of our approach compared to traditional systems.

12.3. What we have proved

The constructions described in the previous section allowed us to make quantitative measurements that support the validity of our approach: specifically, that extensible file properties are feasible, that a special-purpose database is the appropriate mechanism, and that dramatic performance improvements result from applying our approach to several real-world problems.

That the code to implement our designs could be written was not in question, but the existence of our implementation is a "proof of concept'*: programming an extensible-property file system need not require an inordinate amount of labor, nor an unreasonable amount of memory at run-time. More important, the performance of our system, on those few problems at which existing file systems excel, is not significantly worse than a typical existing system. For this small decrease in performance, we get a major increase in function.

We demonstrated that, on problems requiring property-based search, or those that could be expressed that way, our approach improves performance by several orders of magnitude. It converts linear-time problems to logarithmic-time problems, and so scales much better than traditional approaches.

Finally, we have shown that for this application a special-purpose database gives performance superior to that of a general-purpose database, if the special-purpose database is designed to take advantage of domain-specific knowledge. Because the updates and queries our database must support are severely restricted by the problem domain, we used these restrictions to design efficient caching schemes, and limit functionality to what is really necessary.

12.4. What remains to be seen

The ultimate test of the ideas in this thesis is their acceptance by users of a real file system. Acceptance depends in turn on two points: is our approach really useful, and are the performance tradeoffs appropriate?

The utility of extensible properties is contingent on the availability of applications that use them. If the only use of extensibility is for users to attach idiosyncratic annotations to their files, then the potential of file properties to support "smarter" applications will be unrealized. One specific application of extensibility is an archival file server that could encapsulate intrinsic properties of various file systems in a heterogeneous network.

There are two interesting performance tradeoffs that must be tested in a real system. First is

References

[Coll84] William Collins and Stephen Miller.
MSS Generic Model: A Beginning.
In Karen Friedman (editor), *The Mass Storage System Spectrum: A Study in Extremes*, pages 3-9. IEEE Computer Society, Silver Spring, MD, June, 1984.

[Come79] Douglas Comer.
The Ubiquitous B-tree.
ACM Computing Surveys 11:121-137, 1979.

[Corb63] F. J. Corbato.
The Compatible Time-sharing System: a programmer's guide.
MIT Press, Cambridge, MA., 1963.

[Dala81] Y. K. Dalai and R. S. Printis.
48-bit absolute internet and ethernet host numbers. .
In *Proc. 7th Data Communications Symposium*, pages 240-245. ACM/IEEE, October, 1981.
Published as *Computer Communication Review* 11(4).

[Dala86] Yogen Dalai.
Private communication.
1986.

[Dale65] R. C. Daley and P. G. Neumann.
A general purpose file system for secondary storage.
In *Proc. Fall Joint Computer Conference*, pages 213-229. AFDPS, September, 1965.

[Denn82] Dorothy Denning.
Cryptography and Data Security.
Addison-Wesley, Reading, Mass., 1982.

[Denn68] Peter J. Denning.
The Working Set Model for Program Behaviour.
Communications of the ACM 11(5):323-333, May, 1968.

[Denn85] Peter J. Denning.
The Science of Computing: What is computer science?
American Scientist 73(1): 16-19, January-February, 1985.

[DeWi85] D. J. DeWitt.
Benchmarking Database Systems: Past Efforts and Future Directions.
IEEE Computer Society Database Engineering Bulletin 8(l):2-9, March, 1985.

[Digi78] *VAX-11 Software Handbook.*
Digital Equipment Corporation, Maynard, MA., 1978.

[Digi80] *TOPS-20 User's Guide.*
Digital Equipment Corporation, Maynard, MA., 1980.
Form No. AA-4179C-TM.

[Digi83] *VAX/VMS I/O User's Guide, Volume 1.*
Digital Equipment Corporation, Maynard, MA., 1983.

[Dion80] J. Dion.
The Cambridge file server.
Operating Systems Review 14(4):26-35, October, 1980.

[Enge68] D. C. Engelbart and W. K. English.
A research center for augmenting human intellect.
In *Proc. Fall Joint Computer Conference*, pages 395-410. AFIPS, September, 1968.

[Feld79a] Jerome A. Feldman.
High-level programming for distributed computing.
Communications of the ACM 22(6):353-368, June, 1979.

[Feld79b] Stuart I. Feldman.
Make — A Program for Maintaining Computer Programs.
Software—Practice and Experience 9(4):255-265, April, 1979.

[Fras79] A. G. Fraser.
DATAKIT: A modular network for synchronous and asynchronous traffic.
In *Proc. International Conference on Communications*, pages 20.1.1-20.1.3. June, 1979.

[Free84] Gordon George Free.
File Migration in a Unix Environment.
Technical Report UIUCDCS-R-84-1196, Univ. of Illinois, Urbana-Champaign, Dept. of Computer Science, 1984.

[Garf85] Simson L. Garfinkel and J. Spencer Love.
A File System for Write-Once Media.
September, 1985.
In preparation.

[Gold83] A. Goldberg and D. Robson.
Smalltalk-80: The Language and its Implementation.
Addison-Wesley, 1983.

[Gold80] Ira P. Goldstein and Daniel G. Bobrow.
A Layered Approach to Software Design.
Technical Report CSL-80-5, Xerox Palo Alto Research Center, December, 1980.

[Gold81] Ira P. Goldstein and Daniel G. Bobrow.
An Experimental Description-Based Programming Environment: Four Reports.
Technical Report CSL-81-3, Xerox Palo Alto Research Center, March, 1981.

[Grah82] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick.
gprof: a Call Graph Execution Profiler.
In *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*, pages 120-126. ACM SIGPLAN, June, 1982.

[Guru84] Anura Gurugé.
SNA: Theory and Practice.
Pergamon Infotech Ltd., Maidenhead, Berkshire, England, 1984.

References

[Hall80] D. E. Hall, D. K. Scherrer, and J. S. Sventek.
A virtual operating system.
Communications of the ACM 23(9):495-502, September, 1980.

[Hans82] Paul M. Hansen, Mark A. Linton, Robert N. Mayo, Marguerite Murphy, and David Patterson.
A Performance Evaluation of the Intel iAPX 432.
Computer Architecture News 10(4):17-26, June, 1982.

[Hawt85] Paula Hawthorn.
Variations on a Benchmark.
IEEE Computer Society Database Engineering Bulletin 8(1):19-28, March, 1985.

[Herl82] M. Herlihy and B. Liskov.
A value Transmission Method for Abstract Data Types.
ACM Transactions on Programming Languages and Systems 4(4):527-551, October, 1982.

[Holl81] Elmar Holler.
The National Software Works.
In Butler W. Lampson, M. Paul, and H. J. Siegert (editors), *Distributed Systems — Architecture and Implementation: An Advanced Course*, pages 421-445. Springer-Verlag, 1981.

[IBM79] *Using VSE/VSAM Commands and Macros.*
IBM, 1979.
No. SC24-5144-1.

[IEEE85] *A Proposed Standard for Binary Floating-Point Arithmetic.*
IEEE, 1985.
Proposed Standard number P754.

[Inte84] *ISO Transport Protocol Specification.*
International Standards Organization, 1984.
Available as RFC905 from Network Information Center, SRI International.

[Isra78] J. E. Israel, J. Mitchell and H. Sturgis.
Separating data from function in a distributed file system.
In *Operating Systems: theory and practice: Proc. 2nd International Symposium on Operating Systems.* IRIA, North-Holland, New York, October, 1978.

[Kimb81] S. R. Kimbleton and P. Wang.
Applications and protocols.
In Butler W. Lampson, M. Paul, and H. J. Siegert (editors), *Distributed Systems — Architecture and Implementation: An Advanced Course*, pages 308-370. Springer-Verlag, 1981.

[Kron85] Nancy P. Kronenberg, Henry M. Levy, and William D. Strecker.
VAXclusters: A Closely-Coupled Distributed System.
1985.
To appear in *ACM Transactions on Computer Systems*; abstract in *Proc. 10th Symposium on Operating Systems Principles.*

[Lamp81] Butler W. Lampson.
Atomic transactions.
In Butler W. Lampson, M. Paul, and H. J. Siegert (editors), *Distributed Systems — Architecture and Implementation: An Advanced Course*, pages 246-265. Springer-Verlag, 1981.

[Lamp83] Butler W. Lampson and Eric E. Schmidt.
Practical Use of a Polymorphic Applicative Language.
In *Proc. 10th Symposium on Principles of Programming Languages*, pages 237-255. January, 1983.

[Lant85] Keith A. Lantz, Judy L. Edighoffer, and Bruce L. Hitson.
Towards a Universal Directory Service.
In *Proc. 4th Symposium on Principles of Distributed Computing*, pages 250-260. ACM, August, 1985.

[Leff84] Sam Leffler, Mike Karels, and M. Kirk McKusick.
Measuring and Improving the Performance of 4.2BSD.
In *Proc. Summer USENIX Conference*, pages 237-252. June, 1984.

[Lint83] Mark A. Linton.
Queries and views of programs using a relational database system.
PhD thesis, University of California, Berkeley, 1983.
UCB/CSD Technical Report 83/164.

[Lisk77] Barbara H. Liskov, A. Synder, R. Atkinson, and C. Schaffert.
Abstraction mechanisms in CLU.
Communications of the ACM 20(8):564-576, August, 1977.

[Lisk83] Barbara Liskov and Robert Scheifler.
Guardians and Actions: Linguistic Support for Robust Distributed Programs.
ACM Transactions on Programming Languages and Systems 5(3):381-404, July, 1983.

[LISP77] The LISP Machine Group: Alan Bawden, Richard Greenblatt, Jack Holloway, Thomas Knight, David Moon, and Daniel Weinreb.
LISP Machine Progress Report.
Memo 444, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1977.

[Low80] J. R. Low.
Name-type-value (NTV) protocol draft proposal.
Technical Report 73, Department of Computer Science, University of Rochester, July, 1980.

[Maju84] Shikharesh Majumdar.
Locality and File Referencing Behaviour: Principles and Applications.
Technical Report 84-14, University of Saskatchewan Computational Science Department, August, 1984.

[McCa62] John McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin.
LISP 1.5 Programmer's Manual.
The MIT Press, Cambridge, Massachusetts, 1962.

References

[McCa82] John McCarthy.
Private communication.
1982.

[McKu85] M. Kirk McKusick, Mike Karels, and Sam Leffler.
Performance Improvements and Functional Enhancements in 4.3BSD.
In *Proc. Summer USENIX Conference*, pages 519-531. June, 1985.

[Meyr82] Norman Meyrowitz and Andries Van Dam.
Interactive Editing Systems.
ACM Computing Surveys 14(3):321-416, September, 1982.

[Mitc82] J. G. Mitchell and J. Dion.
A comparison of two network-based file servers.
Communications of the ACM 25(4):233-245, April, 1982.
Presented at the 8th Symposium on Operating Systems Principles, ACM, December 1981.

[Mogu81] Jeffrey Mogul.
Leaf and Sequin Protocols.
1981.
Department of Computer Science, Stanford University, unpublished technical report.

[Nels81a] Bruce Jay Nelson.
Remote Procedure Call.
Technical Report CSL-81-9, Xerox Palo Alto Research Center, May, 1981.

[Nels65] T. H. Nelson.
A File Structure for the Complex, the Changing, and the Indeterminate.
In *Proc. National Computer Conference*, pages 84-100. ACM, August, 1965.

[Nels81b] Ted Nelson.
Literary Machines.
P. O. Box 128, Swarthmore, PA. 19081, 1981.

[Nowi85] William I. Nowicki.
Partitioning of Function in a Distributed Graphics System.
PhD thesis, Stanford University, 1985.

[Oppe83] Derek C. Oppen and Yogen K. Dalal.
The Clearinghouse: A decentralized agent for locating named objects in a distributed environment.
ACM Transactions on Office Information Systems 1(3):230-253, July, 1983.

[Patt85] D. A. Patterson.
Reduced instruction set computers.
Communications of the ACM 28(1):8-21, January, 1985.

[Pope81] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel.
LOCUS: A network transparent, high reliability distributed system.
In *Proc. 8th Symposium on Operating Systems Principles*, pages 169-177. ACM, December, 1981.
Published as *Operating Systems Review* 15(5).

[Post81a] Jon Postel.
Internet Protocol.
RFC 791, Network Information Center, SRI International, September, 1981.

[Post81b] Jon Postel.
Transmission Control Protocol.
RFC 793, Network Information Center, SRI International, September, 1981.

[Powe83] Michael L. Powell and Barton P. Miller.
Process migration in DEMOS/MP.
In *Proc. 9th Symposium on Operating Systems Principles*, pages 110-119. ACM, October, 1983.
Published as *Operating Systems Review* 17(5).

[Reed81] David Reed and Liba Svobodova.
SWALLOW: A distributed data storage system for a local network.
In A. West and P. Janson (editors), *Local Networks for Computer Communications*, pages 355-373. North-Holland, 1981.

[Reed86] Sandra R. Reed.
Getting to Know Your Spreadsheet.
Personal Computing 10(2):134, February, 1986.

[Reid80] Brian K. Reid, Janet H. Walker.
Scribe User's Manual.
Unilogic, Inc, Pittsburgh, PA., 1980.

[Ritc78] D. M. Ritchie and K. Thompson.
The UNIX timesharing system.
The Bell System Technical Journal 57(6):1905-1929, July/August, 1978.

[Robe79] G. Robertson, D. McCracken, and A. Newell.
The ZOG approach to man-machine communication.
CMU-CSD 79-148, Department of Computer Science, Carnegie-Mellon University, October, 1979.

[Roch75] M. J. Rochkind.
The Source Code Control System.
IEEE Transactions on Software Engineering SE-1(4):364-370, December, 1975.

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

[Sand85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon.
Design and Implementation of the Sun Network Filesystem.
In *Proc. Summer USENIX Conference*, pages 119-130. 1985.

[Saty85] M. Satyanarayanan, John H. Howard, David A. Nichols, Robert N. Sidebotham, Alfred Z. Spector, and Michael J. West.
The ITC Distributed File System: Principles and Design.
In *Proc. 10th Symposium on Operating Systems Principles*, pages 35-50. ACM, December, 1985.
Published as *Operating Systems Review* 19(5).

References

[Schm82] Eric E. Schmidt
Controlling Large Software Development in a Distributed Environment
PhD thesis, University of California, Berkeley, December, 1982.
Xerox PARC Technical Report CSL-82-7.

[Schr85] Michael D. Schroeder, David K. Gifford, and Roger M. Needham.
A Caching File System for A Programmer's Workstation.
In *Proc. 10th Symposium on Operating Systems Principles*, pages 25-34.
ACM, December, 1985.
Published as *Operating Systems Review* 19(5).

[Shaw77] Mary Shaw and William Wulf.
Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators.
Communications of the ACM 20(8), August, 1977.

[Sing85] Kamaljit Singh.
On Improvements to Password Security.
Operating Systems Review 19(l):53-59, January, 1985.

[Smit78] Alan Jay Smith.
Long Term File Migration.
Technical Report, Department of Electrical Engineering and Computer Science, University of California — Berkeley, 1978.

[Smit85] David E. Smith.
Controlling Inference.
PhD thesis, Stanford University, August, 1985.

[Spec82] Alfred Z. Spector.
Performing remote operations efficiently on a local computer network.
Communications of the ACM 25(4):246-260, April, 1982.
Presented at the 8th Symposium on Operating Systems Principles, ACM, December 1981.

[Spec83] Alfred Z. Spector and Peter M. Schwarz.
Transactions: A construct for reliable distributed computing.
Operating Systems Review 17(2): 18-35, April, 1983.

[Stal81] Richard M. Stallman.
EM ACS: The Extensible, Customizable, Self-Documenting Display Editor.
SIGPLAN Notices 16(6): 147-156, June, 1981.

[Stan84] *A Reference Guide to SOCRATES: The Online Library Catalog of Stanford University.*
Stanford University Libraries, 1984.

[Ston76] M. Stonebraker.
The design and implementation of INGRES.
ACM Transactions on Database Systems 1(3): 189-222, September, 1976.

[Ston81] M. Stonebraker.
Operating system support for database management.
Communications of the ACM 24(7):412-418, July, 1981.

[Stri77] Edward Stritter.
File Migration.
PhD thesis, Stanford University, December, 1977.

[Su82] Zaw-Sing Su and Jon Postel.
The Domain Naming Convention for Internet User Applications.
RFC 819, Network Information Center, SRI International, August, 1982.

[Sun85] *Networking on the Sun Workstation.*
Sun Microsystems, Inc., Mountain View, Ca., 1985.

[Svob81] Liba Svobodova.
A reliable object-oriented data repository for a distributed computer system.
In *Proc. 8th Symposium on Operating Systems Principles*, pages 47-58.
ACM, December, 1981.
Published as *Operating Systems Review* 15(5).

[Svob84] Liba Svobodova.
File Servers for Network-Based Distributed Systems.
ACM Computing Surveys 16(4):353-398, December, 1984.

[Swin79] D. Swinehart, G. McDaniel, and D. Boggs.
WFS: A simple shared file system for a distributed environment.
In *Proc. 7th Symposium on Operating Systems Principles*, pages 9-17.
ACM, December, 1979.
Published as *Operating Systems Review* 13(5).

[Symb85] *Reference Guide to Streams, Files, and I/O.*
Symbolics, Inc, Cambridge, MA., 1985.

[Teit81] Tim Teitelbaum and Thomas Reps.
The Cornell Program Synthesizer: A Syntax-Directed Programming Environment.
CACM 24(9):563-573, September, 1981.

[Teit75] Warren Teitelman.
INTERLISP reference manual.
Xerox Palo Alto Research Center, Palo Alto, California, 1975.

[Thac82] C. P. Thacker, E. M. McCreight, B. W. Lampson, R. F. Sproull, and D. R. Boggs.
Alto: A personal computer.
In D. P. Siewiorek, C. G. Bell, and A. Newell (editors), *Computer Structures: Principles and Examples*, pages 549-572. McGraw-Hill, 1982.

[Thei86] Marvin Theimer.
Preemptable Remote Execution Facilities for a Distributed Computer System.
PhD thesis, Stanford University, 1986.
In preparation.

[Thom85] Mary R. Thompson, Robert D. Sansom, Michael B. Jones, and Richard F. Rashid.
Sesame: The Spice File System.
Technical Report CMU-CS-85-172, Department of Computer Science, Carnegie-Mellon University, December, 1985.

References

[Tich82] Walter F. Tichy.
Design, Implementation, and Evaluation of a Revision Control System.
In *Proc. 6th International Conference on Software Engineering.* IEEE, Tokyo, September, 1982.

[Tich85] Walter F. Tichy.
Smart Recompilation.
In *Proc. 12th Symposium on Principles of Programming Languages*, pages 236-244. January, 1985.

[Univ85] *Univac Series 1100 Executive System, EXEC level 39R3, Programmer Reference.*
UP-4144.32 edition, Sperry Corporation, Blue Bell, Pennsylvania, 1985.

[Wats81] R. W. Watson.
Distributed system architecture model.
In Butler W. Lampson, M. Paul, and H. J. Siegert (editors), *Distributed Systems — Architecture and Implementation: An Advanced Course*, pages 10-43. Springer-Verlag, 1981.

[Wilk64] M. V. Wilkes.
A Programmer's Utility Filing System.
Computer 7:180-184, October, 1964.

[Wilk79] M. V. Wilkes and R. M. Needham.
The Cambridge CAP Computer and Its Operating System.
North-Holland/Elsevier, 1979.

[Wino75] Terry Winograd.
Frame Representation and the Declarative-Procedural Controversy.
In Daniel G. Bobrow and Allan Collins (editors), *Representation and Understanding: Studies in Cognitive Science*, pages 185-210. Academic Press, New York, 1975.

[Wulf81] W. A. Wulf, R. Levin, and S. P. Harbison.
HYDRA/C.mmp: An Experimental Computer System.
McGraw-Hill, 1981.

[Xero75] Xerox Palo Alto Research Center Learning Research Group.
Personal Dynamic Media.
Technical Report SSL 76-01, Xerox Palo Alto Research Center, 1975.

[Xero79] *Alto operating system reference manual.*
Xerox Palo Alto Research Center, Palo Alto, California, 1979.

[Xero81] Xerox Office Products Division.
Courier: The Remote Procedure Call Protocol.
Xerox System Integration Standard XSIS 038112, Xerox, December, 1981.

[Xero83] *INTERLISP reference manual.*
Xerox Special Information Systems, Pasadena, California, 1983.

[Xero85a] *NoteCards 1.2 Reference Manual.*
Xerox Special Information Systems, Pasadena, California, 1985.

[Xero85b] *Xerox File Protocol*
XNSS 108507 edition, Xerox Network Systems Institute, Palo Alto, California, 1985.
(Draft edition).