Palo Alto Research Centers

Information Storage in a Decentralized Computer System

David K. Gifford

XEROX

Information Storage in a Decentralized Computer System

by David K. Gifford

CSL-81-8 June 1981; Revised March 1982

Abstract: See page x.

This report is a slightly revised version of a dissertation submitted to the Department of Electrical Engineering and the Committee on Graduate Studies of Stanford University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

CR categories: 4.33, 4.35, 3.73

Key words and phrases: file system, information storage, reference, transaction, location, replication, suite, representative, quorum, reconfiguration, protection, cryptographic sealing, key, seal, unseal.

XEROX

Xerox Corporation
Palo Alto Research Centers
3333 Coyote Hill Road
Palo Alto, California 94304

Contents

Figures

Tables

Preface

This paper is organized so that it can be used in several ways. It can of course be read in its entirety for a treatment of the general problem of decentralized information storage. It is also possible to read an isolated chapter for a treatment of a single topic, such as protection. Each chapter begins with an outline of its organization and ends with a summary of its contents. If the reader wishes to look at a chapter in isolation I would also suggest that they read Sections 1.3 and 3.1.

There is a comprehensive index at the end of the paper. The index is useful to locate references to a specific concept. It also indexes the program text, and thus should help a reader find the definition of a function or a type.

I have tried to present enough detail so a reader can easily implement the ideas I discuss. However, it is possible to understand the essence of the ideas without reading program text. On a first reading, a casual reader should probably skip sections titled implementation or refinement.

I would like to thank my patient advisers, Dr. Butler Lampson and Prof. Susan Owicki, for their help and good advice over the past four years. In addition to my advisers, Peter Deutsch, Jim Gray, Martin Haeberli, Prof. Larry Manning, Gene McDaniel, Alfred Spector, Larry Stewart, and Howard Sturgis took the time to carefully read chapters of this paper and suggest improvements. I would also like to thank Bob Taylor for making it possible for me to do this work at the Xerox Palo Alto Research Center. The work benefited greatly from daily interactions with colleagues at Xerox.

The Fannie and John Hertz Fellowship that I held while a graduate student gave me the freedom to pursue this research. The research was supported in part by the Fannie and John Hertz Foundation, and by the Xerox Corporation.

This work is dedicated to my parents.

Abstract

This paper describes an architecture for shared information storage in a decentralized computer system. The issues that are addressed include: naming of files and other objects (naming), reliable storage of data (stable storage), coordinated access to shared storage (transactional storage), location of objects (location), use of multiple copies to increase performance, reliability and availability (replication), dynamic modification of object representations (reconfiguration), and storage security and authentication (protection).

A complete model of the architecture is presented, which describes the interface to the facilities provided, and describes in detail the proposed mechanisms for implementing them. The model presents new approaches to naming, location, replication, reconfiguration, and protection. To verify the model, three prototypes were constructed, and experience with these prototypes is discussed.

The model names objects with variable length byte arrays called references. References may contain location information, protection guards, cryptographic keys, and other references. In addition, references can be made indirect to delay their binding to a specific object or location.

The replication mechanism is based on assigning votes to each copy of a replicated object. The characteristics of a replicated object can be chosen from a range of possibilities by appropriately choosing its voting configuration. Temporary copies can be easily implemented by introducing copies with no votes.

The reconfiguration mechanism allows the storage that is used to implement an object to change while the system is operating. A client need not be aware that an object has been reconfigured.

The protection mechanism is based on the idea of sealing an object with a key. Sealed objects can only be unsealed with an appropriate set of keys. Complex protection structures can be created by using such operators as Key-Or and Key-And. The protection mechanism can be employed to create popular protection mechanisms such as capabilities, access control lists, and information flow control.

Chapter 1: Method and Principles

1.1 Introduction

Communication is an essential part of our basic need to cooperate and share with one another. We have been given the freedom to have distant friends and increased knowledge about our world by advances in communication technology such as the post office, the telegraph, and the telephone. Computer systems promise to be another such advance. Large scale community information systems are likely to play a major role in our future ability to create, organize, process, store, and share information.

It was once thought that the problem of building large computer systems was that of building large computers. It is now clear that this is not the case. Instead of employing a single computer, future large scale computer systems will be composed of thousands, or even millions, of computers.

The goal of this research is to demonstrate an information storage system architecture that can be used to integrate a collection of computers. By integrate we mean that the architecture permits information to be easily exchanged between the users of the system. We are not suggesting that information storage be centralized. Rather, we propose to organize the storage facilities that would normally exist in a collection of computers into a single decentralized system.

The information storage architecture we present is intended to create a foundation for shared applications. Example applications include office information systems, programming environments, and data base systems. The principles of our architecture are best characterized by the desire to provide fundamental storage facilities that can be flexibly adapted to a wide range of uses.

1.2 Method

Computer system research is more than the invention of new algorithms; part of the work lies in the synthesis of a collection of ideas into a single package. Furthermore, it is important that a synthesis be faithful to a single set of coordinated architectural principles. Conceptual integrity keeps complex interactions from making the system intractable as it increases in size and function. A clear statement of principal design decisions is central to the overall success of a large system. The importance of these ideas have been demonstrated by [Belady and Lehman 77] and [Brooks 75].

We suggest a four stage process for system creation that is intended to promote these concepts. The idea of the process is to emphasize the importance of asking fundamental questions early in the life of a system, and to postpone secondary decisions. In order, the four stages are:

1. Define the system's *architectural principles.* The architectural principles of a system are a set of primary design decisions that consider technical feasibility [Liddle 76]. These decisions serve to define and delimit the scope of a system. Furthermore, they allow for orderly growth by providing a single conceptual framework that can accommodate extensions in system size and function.

   For example, what is the nature of the system? Is it intended to provide a general purpose computing environment? Or is going to be used exclusively for electronic mail? How large

will the system be? Are different instances of the system going to be interconnected? What are the capacity and reliability requirements of the system? What are the services that the system will offer? How will these services be presented to a client? What are the requirements for accounting? For protection?

2. Formulate a *system model.* A system model is a design for the system in line with its architectural principles. A model describes the system's interfaces and mechanisms in enough detail that it is possible to reason about the correctness of key algorithms. When the system is constructed, the system model is used as a pattern.

3. Implement the system. A *system implementation* is a concrete set of hardware and software components that realize a system model. The implementation of a system normally starts by making a plan for its construction, testing, and documentation. Naturally, there can be several implementations of a system model. This is important, as over the life of a system new implementations of parts of a system will cause new and old components to coexist.

4. Plan a *system configuration.* A system configuration is an installed set of components from a system implementation. A system implementation represents a wide spectrum of capacity, reliability, availability, and performance possibilities; a configuration reflects the decisions made to meet specific needs.

Our research has been organized according to these stages.

The remainder of this chapter treats the architectural principles of our system and their background. We discuss how time-sharing systems, personal computing systems, and computer networks have influenced our goals.

The next six chapters describe our system model by successive refinement. Chapter 2 defines the environment of the system model. Chapter 3 presents a simple storage system. This storage system would be ideal if one made the following assumptions:

1. Files, volumes, and other objects never move.

2. It is never necessary to improve the reliability, availability, or performance characteristics of storage devices.

3. It is never necessary to change the storage that is used to store an object.

4. People are perfectly trustworthy and there is no need for protection.

Chapters 4 through 7 remove these assumptions. We add location (Chapter 4), replication (Chapter 5), reconfiguration (Chapter 6), and protection (Chapter 7) to describe a practical system.

Chapter 8 discusses system implementation and configuration. Three prototypes were built to test the validity of our system model, and experience with these prototypes is discussed. In view of these prototypes, we outline our expectations about full scale implementations of the system. System configuration is discussed briefly, but it is not thoroughly explored.

Chapter 9 concludes the paper with a summary of the major ideas introduced in the paper and a review of how we have achieved the architectural principles we set forth.

1.3 System Description

This paper includes a large amount of program text. The intent of the program text is to provide the reader with a detailed understanding of the system model. Considered collectively, the program text is an implementation of the system model. The program text is also intended to be used as a pattern for full-scale implementations. Thus, the code that we present is called a *model implementation.* The functions and types of the model implementation are indexed at the back of the paper.

The system model is described in EL ("Exposition Language"), a dialect of Lisp 1.5 [McCarthy et al. 62]. Lisp was chosen because it includes an Eval function, which allowed us to define the semantics of operations executed at remote processors. In addition, Lisp lends itself to the transformation of objects to byte strings and back again (see the descriptions of Encode and Decode in the appendix). Appendix A should provide enough information to enable the reader to understand the code in the paper. Much of the technical content of the paper is contained in the program text, and thus we suggest that the reader take the time now to read Appendix A.

To help the reader understand EL we present a short example program fragment. The fragment shown below is not intended to be useful, but it does demonstrate some key EL constructs. We start by defining three record types: Person, Experience, and Experienced-Person. Experienced-Person is a derived record type that contains the fields and types of Person and Experience. The function Open-Person creates a new class that services the operations Name and Parent. Open-Person takes a person record and a person class as inputs. They respectively represent a person and that person's parent. When a Name request is sent to a class, Person-Name is invoked, and the name of the person is returned. When a Parent request is sent to a class, Parent-Name is invoked. Parent-Name gets the name of the class' parent from its superclass, which was set when the class was created.

Every time that we create a class we include a comment that describes the class' *instance variables.* For example, Open-Person creates a class, and Person-Name and Parent-Name can use the instance variable *name.* The value of instance variables persist over class activations.

```
Person ← Record[name: Byte-Array];
Experience ← Record[years: Integer];
Experienced-Person ← Extend[Person, Experience];

Create-Person[name: Byte-Array / p: Person] ← Prog[ [];
        p ← Create[Person];      p.name ← name;
        Return[p];
        ];

Open-Person[p: Person, parent: Person-Class / c: Person-Class] ← Prog[
        [name: Byte-Array];
        -- copy name
        name ← p.name;
        -- create a new class
        c ← Create-Class[List[
                'Name, 'Person-Name,
                'Parent, 'Parent-Name], parent];
```

```
        -- Instance variables: name
        Return[c];
        ];

Person-Name[/n: Byte-Array] ← Prog[ []];
        -- Person-Name is evaluated in the environment of Open-Person
        Return[name];
        ];

Parent-Name[/n: Byte-Array] ← Prog[ []];
        -- ask superclass for its name
        n ← superclass | Name[];
        Return[n];
        ];


frank-class ← Open-Person[Create-Person["Frank"], NIL];
alfred-class ← Open-Person[Create-Person["Alfred"], frank-class];
-- dad will be "Frank"
dad ← alfred-class | Parent[];
```

We observe the following stylistic contentions in program text. Variables always begin with lower-case letters. Function names and record types are capitalized. Whenever a new object is introduced, we follow the same order of presentation as we did in our example. First, we introduce the operation to create an object instance. Second, we describe an "open" function that returns a class that will service an object instance. Third, we present the functions that actually implement the class' operations.

Let us define some terms that we will use repeatedly throughout this paper. A *client* is a program that uses the facilities we describe, and a *user* is a human being that interacts with a client. The *reliability* of a system is a measure of the probability that the system will malfunction, and a system's *availability* is a measure of the probability that it will be operational when it is needed.

1.4 Background

Early in the 1960's time-sharing was introduced as a way of providing the illusion of a personal computer to aid in program debugging. Time-sharing systems turned out to provide another benefit that was not originally anticipated. Users found they could easily share information that was stored in a time-sharing system. Sharing proved to be easy because it was as if a single file cabinet simultaneously existed in every user's office. Items placed in one file cabinet immediately appeared in the rest of the file cabinets. The facilities for information sharing provided by time-sharing soon found use in large collaborative software projects.

As time-sharing matured, sharing was recognized as a basic facility, and made correspondingly convenient. The CTSS system [Corbato et al. 62] pioneered multiple access computers, and provided a simple shared file system. Based on this experience, Multics [Corbato et al. 72] extended its file system to include a tree-structured naming system and advanced protection facilities.

Late in the 1970's hardware became inexpensive enough that users could be provided their own computers [Thacker et al. 79]. Placing a large amount of computational power at the man-machine

interface dramatically improved its character. For example, graphics could be introduced, adding an important element to the domain of things that computers could be used for. Communication networks [Metcalfe and Boggs 76] allowed the users of personal computers to share resources such as high-speed printers. However, sharing was not made as convenient as it was in time-sharing systems.

This research proposes to combine the success of time-sharing's shared storage and the success of personal computing's man-machine interface into a single system. Unfortunately, it is not sufficient to simply add conventional storage to personal computing systems. In a decentralized environment such problems as coordination, protection, reliability, availability, and performance become much more complicated. For example, in a decentralized system when information is transferred between computers over an insecure channel it must be encrypted to provide protection.

Furthermore, people's expectations have properly increased. Computers are being used for an increasingly diverse spectrum of applications, and many computer users are no longer computer professionals. Our understanding of these requirements is reflected in the principles of our architecture.

A number of systems have been built that share our primary goal of integrating a collection of computers with a shared information storage system. These systems fall into three broad categories.

1. Existing time-sharing systems have been modified to access remote files. The RSEXEC [Thomas 73] system was an early attempt to join TENEX systems in this manner, and the RSEXEC approach was later adopted by the National Software Works [Forsdick et al. 77]. The Locus project [Popek et al. 81] at UCLA has integrated a number of UNIX systems in a similar manner. Locus includes facilities for mediating concurrent access to information, and there are plans to incorporate replicated data as well. These systems all have major restrictions that are rooted in their time-sharing origins. In addition, they have as a general rule adopted ad hoc solutions to the intrinsic and environmental problems they faced. Thus, they do not provide a general framework of the sort we propose.

2. Data base systems have been extended to operate on several computers that are connected by a network. Examples of such systems are CICS ISC [IBM 80a] and Tandem Computer's Encompass [Tandem 81]. These systems are intended to provide a specialized service. In addition, they have not provided general solutions to many of the problems that we consider.

3. File servers have been constructed for local computer networks, and these file servers have been used by client computers for shared storage. WFS, and its successor, IFS, are two such file servers [Swinehart et al. 79]. A file server at the University of Cambridge has been successfully used as the only storage service of a time-sharing system [Dion 80]. The Xerox Distributed File System [Israel et al. 1978] provides facilities for guaranteeing information consistency across file servers, and sophisticated facilities for failure recovery. These file servers are more general than the time-sharing based efforts, and motivated the system we propose. However, the scope of these servers is limited, and they do not address many of the problems that we consider.

1.5 Architectural Principles

The following twelve architectural principles describe and delimit the scope of our ideal storage system. We will return to this list at the end of the paper to review how they have been satisfied.

1. The system should behave in a well defined way. In a large system design there are many opportunities for selecting a mechanism that works most of the time. Such a mechanism can only be employed in conjunction with a backup mechanism that is expected to work all of the time. For example, certain existing systems will undetectably malfunction in unusual circumstances. We will not consider such designs.

2. The system should provide a basic storage service. The basic unit of storage should be the file, an uninterpreted array of bytes. Read and write primitives should be provided to access files. The notion of a volume should also be provided to model storage media. Files are created and stored on volumes.

3. Storage should be resilient to expected failures. From time to time hardware errors, system errors, or operator errors will occur. The storage system should expect such errors, and recover from them without information loss. Furthermore, if unexpected errors occur, the system should indicate that storage has been damaged instead of providing incorrect information.

4. Files, volumes, and other objects should be named with unambiguous low-level names. The storage system should not anticipate how clients might use these names or what naming environments will be presented to users.

5. The system should mediate concurrent access to storage to ensure consistency.

6. The system should be decentralized, and the location of storage system objects should be hidden from clients. The system should also allow clients to discover where objects are located.

7. The system should allow modular expansion. The storage capacity of the system should not be limited by any design decision, nor should the design intrinsically limit the number of users that the system can support.

8. It should be possible to improve the performance, reliability, and availability of the storage system by keeping multiple copies of selected storage system objects. This principle includes the idea of making temporary copies of objects for rapid access.

9. It should be possible to reconfigure the system while it is operating. Reconfiguration involves changing the storage resources that are used to implement a storage system object.

10. A mechanism for information secrecy and authentication should be provided. The mechanism should be general enough that clients can use it to implement a variety of protection policies. No one should be able to circumvent the protection mechanism. For example, system administrators should not be able to access information that they are not authorized to see.

11. It should be possible to construct derived volumes by extending existing volumes with

replication, reconfiguration, and protection structures. Files created on derived volumes will use the volume's structure as a template. For example, it should be possible to create a volume R from three other volumes A, B, and C. When a file F is created on R, copies of F will be automatically maintained on A, B, and C. This allows popular classes of storage to be directly represented in the system as volumes. Thus, clients may choose to ignore what facilities are being used to provide the storage they use.

12. A client should be able to select the resources that it uses in such a way that its processor can remain autonomous from the rest of the system.

1.6 Summary

A new information storage system was proposed. It is intended to be the foundation of diverse applications such as office information systems, programming environments, and data base systems. The design of the system was divided into four stages: definition of architectural principles, formulation of a system model, system implementation, and system configuration. The paper is structured according to these stages. The system's architectural principles and their background were then introduced. Chapter 2 begins our consideration of a system model to realize these principles.

Chapter 2: Environment

The first step in defining the system model is to outline its presumed environment. This chapter reviews the characteristics of the hardware components we use, introduces the concepts of stable storage and unique identifiers, and shows how to perform reliable function evaluation at a remote processor. The material in this chapter represents an integration of ideas that have been presented before in various forms. Notably [Lampson and Sturgis 79] previously introduced a number of the concepts reviewed here.

2.1 Hardware Components

Three types of hardware components comprise the system: processors, communication channels, and storage devices. In each of the following sections we discuss the characteristics of each type of hardware component. The hardware model presented is abstract to the extent that it only concerns itself with device characteristics that will influence later design decisions.

2.1.1 Processors

A *processor* corresponds to the familiar notion of a stored program digital computer. Processors are the active elements in a system, and they operate independently from one another. Processors only communicate with each other through communication channels. There is absolute protection between processors in the sense that all a malicious processor can do is send messages, which other processors can choose to ignore. Depending on application needs, processors may be connected to a wide variety of peripherals, such as storage devices, bit-map displays, pointing devices, bar-code readers, and laser printers. Although it is not an absolute necessity, we will assume that all processors in the system have the following ideal capabilities.

Every processor is assigned an identifier that is distinct from any other processor's identifier. Processor identifiers can be implemented by including a read-only memory in each processor that contains its identifier. Fixed length processor identifiers offer simplicity, but for some implementations the expansion capability offered by variable length identifiers may be attractive. A processor's identifier can be discovered with the function GetProcessorID.

GetProcessorID[/id: ProcessorID]

GetProcessorID returns the identifier of the processor that the calling process is using.

In addition, it is assumed that a processor can encrypt and decrypt data, and generate true random numbers. Encryption can be implemented in software, but for efficient operation with high security codes it is likely that special purpose hardware will be required. True random numbers are useful for generating hard to guess cryptographic keys. A true random number is not the output of a pseudo-random number generator, but is derived from a truly random process such as thermal noise, shot noise, or radioactive decay.

2.1.2 Communication

Communication and synchronization are accomplished between processors by sending and receiving messages. A message is an uninterpreted array of bytes that is sent to a destination processor. Every processor is assigned an address that includes its processor identifier, so different processors will never be assigned the same address. However, we assume that if a processor is physically moved its address will change.

The following functions define the interface to the communication system. Note that *Processor* includes a processor identifier and fields that are private to the communication system.

Processor ← Extend[Record[id: ProcessorID], PrivateFields];

Send[destination: Processor, message: Byte-Array]

> Send transmits *message* to *destination.*

Receive[/message: Byte-Array]

> Receive delays until a message arrives addressed to this processor, and then returns the incoming message.

GetMyProcessor[/self: Processor]

> GetMyProcessor returns the address of this processor.

The communication system can lose, duplicate, or arbitrarily delay messages. Thus, messages may arrive out of order, more than once, or not at all. We will assume that messages damaged in transit are detected and discarded by the communication system, and will appear to have been lost.

At least one distinguished address, *Broadcast*, is defined by the communication system so that a processor can discover things about its environment and begin communicating with other processors. One or more processors can ask to receive messages addressed to *Broadcast.* How far broadcast messages will propagate in the communication system is implementation dependent. Chapter 4 discusses how this facility is used.

> Broadcast: Processor

Packet switched networks [Metcalfe 73] are well adapted to providing a full connectivity network at a moderate cost, and are an attractive method for implementing the proposed communication system. Local packet switched networks, such as the Ethernet [Metcalfe and Boggs 76], provide high capacity and low delay at low cost for a local area. A treatment of local networks can be found in [Clark et al. 78].

Local networks can be connected together by gateway processors and communication channels to form an internetwork [Boggs et al. 80]. An internetwork retains the performance and cost advantages of a local network, while extending the communication system to accommodate modular growth. In an internetwork, non-local packets are forwarded though a succession of gateways to eventually arrive at their destination.

2.1.3 Storage devices

A *storage device* is a processor peripheral that stores data. Read and write are the two primitive operations that are used to access and update storage devices, respectively. We assume that a

storage device will indicate when it has failed to accurately store data. Failure detection is typically implemented with the help of labels and checksums [Lampson and Sproull 79].

Three fundamental characteristics of storage devices are capacity, latency, and transfer rate. The capacity of a storage device is the maximum number of bytes that it can store on a single medium. We have been careful to state maximum, because the amount of data that can be stored is often a function of the type of media in use. The transfer rate is the maximum number of bytes per second that can be transferred to or from the device. The latency of a storage device is the average amount of time required to read or write information ignoring the time the device is actually transferring data.

Storage devices have four additional important characteristics.

1. Some storage devices are designed for random access to data, and some are designed for serial access. These devices will be called *random access* and *serial access*, respectively. A tape drive is an example of a device designed for serial access to data, and an attempt to use it in a random access mode would result in poor performance. Although the storage capacity of random access devices is increasing, serial access devices may continue to have a role in information storage because of their lower cost.

2. Some storage devices allow data to be read or written, but once data is stored it can not be overwritten. Such a device will be called *write-once*. This property is usually due to the storage media in use, for example a write-once optical disk.

3. Some storage devices do not allow data to be written. Such a device will be called *read-only*. As a safety feature some storage devices can be made temporarily read-only.

4. Some storage devices can record on storage media that are interchangeable between other storage devices of the same type. Such a device will be said to have *removable media*.

2.2 Stable Storage

Storage that is resilient to a set of expected failures is called *stable storage*. From time to time hardware errors, system errors, or human errors will occur. A stable storage system expects such errors, and recovers from them without information loss. Furthermore, if unexpected errors occur, a stable storage system indicates that storage has been damaged instead of providing incorrect information. Expected hardware failures include storage device transfers that malfunction, and information on a storage device that decays and becomes unreadable. Transfers that malfunction include transfers that are in progress when their controlling processor fails.

The write-ahead-log protocol [Gray 78] is widely used to implement stable storage. It implements stable storage by carefully maintaining two copies of data on devices with independent failure modes. [Lampson and Sturgis 79] suggest a similar algorithm.

It is difficult to precisely characterize the reliability of stable storage. Manufacturers give bit error rates for their devices, but these figures do not include catastrophic failures. Examples of catastrophic failures are head crashes on disk drives, media dropped on the floor by operators, and media mistakenly erased by users.

We shall call storage that is not protected against failures *volatile storage*. Stable storage will be used to record long term system state, and volatile storage will be used for intermediate results.

Volatile storage is naturally much more efficient than stable storage because the same precautions do not have to be observed.

2.3 Unique Identifiers

A *unique identifier* is defined to be a value that is distinct from every other unique identifier. We will assume for simplicity that unique identifiers are system generated nonsensical bit strings, but unique identifiers could be client generated and sensible.

All unique identifiers do not have to be the same length. When it is known that a unique identifier is going to be used extensively it may be advantageous to use a shorter identifier. Of course, there are fewer short identifiers than there are long ones.

A common method for generating a unique identifier is to concatenate a processor identifier with a locally unique identifier. A locally unique identifier can be implemented by a counter in stable storage. Whenever a locally unique identifier is required the counter is incremented and returned. An obvious optimization is to withdraw a sequence of locally unique identifiers from stable storage at once to reduce delay. Another technique for generating locally unique identifiers is as follows. At processor initialization time create a variable *nextId* and set it equal to a calendar clock. A calendar clock is a clock that holds the current date and time and thus monotonically increases. Every time a locally unique identifier is requested increment *nextId* by one and ensure that it is less than the calendar clock by pausing if necessary. Now *nextId* is guaranteed to be locally unique. Thus, the second scheme does not require stable storage.

Although theoretically unique identifiers are unique, there is a chance that the unique identifier mechanism could fail and issue duplicate identifiers. Such a failure could result from two processors that were mistakenly assigned the same processor identifier, or from a malicious client. The algorithms we present in many instances check for duplicate unique identifiers. However, to provide a foundation on which to build, we will assume that unique identifiers are in fact unique.

The following function provides unique identifiers:

GetUniqueID[/id: UniqueID]

> GetUniqueID returns a unique identifier. On a single processor subsequent unique identifiers from GetUniqueID are monotonically increasing.

2.4 Reliable Remote Evaluation

2.4.1 Model

Remote form evaluation is the way one processor requests another processor to evaluate a function and return a result. A remote evaluation is a generalization of what is commonly referred to as a remote procedure call. [Spector 80] provides a taxonomy of remote operations and their semantics.

It is possible to provide precise semantics for remote evaluation because evaluation is formally defined with respect to an environment that binds values to free variables and functions. We will assume that remote evaluation is done with respect to a default environment that includes the data types and functions defined in the paper.

To communicate with a remote processor it is first *opened.* Opening a processor results in a

class that will service *Eval* requests. Eval will evaluate an EL form at the remote processor. To make the remote evaluation of forms easier, we restrict forms to be defined by the following grammar:

<form> ::= Function[<form> ... <form>]
<form> ::= <variable>
<form> ::= Quote[<expression>]

where <variable> is a local variable, and <expression> is an unrestricted expression.

Open-Processor[processor: Processor / rp: Processor-Class]

Open-Processor creates a class that will service requests for remote form evaluations at the specified processor. It is possible to specify *Broadcast* as the processor, in which case the first processor to respond to the open request will be selected.

rp: Processor-Class | Eval[form: Form / result: Any]

Assuming there are no processor failures, rp | Eval will evaluate *form* at the processor specified by rp exactly once, returning the result of the evaluation. rp | Eval[x] is equivalent to rp | Eval[Copy[x]]. That is, the evaluation of *form* will have no side effects on the local processor.

rp | Eval hides communication system failures, but it does not mask processor failures. If a remote processor fails while it is evaluating *form*, Error['ProcessorFailure] is returned. rp | Eval is intended to be a low cost method of communication, and thus stable storage was considered to be too expensive to use as an integral part of its implementation. Without stable storage it is impossible to remember the state of an evaluation that is in progress when a processor fails, and thus it is impossible to mask processor failures.

rp | Eval will not return until it has successfully performed a requested evaluation. It will keep trying even if a processor is unavailable. A processor can be unavailable for many reasons, such as hardware failure, software failure, or a problem with the communication system.

rp: Processor-Class | Close[]

Close deactivates a processor class.

If a processor that is participating in a remote evaluation fails one must assume that portions of the evaluation will continue to be in execution for an indefinite time. Imagine that processor A requests processor B to evaluate F, and the evaluation of F on processor B requests that processor C evaluate G. Such a case might arise if F was "write replicated file" and G was "write file copy". Now processor B fails, and processor A requests processor B to evaluate F again. Even if processor B finishes evaluating F successfully on its second try, processor C may still be executing a no longer needed evaluation of G. Such an unchecked evaluation has been called an *orphan* [Nelson 81]. The orphan resulted from the partial evaluation of F, and may modify the state of the system sometime in the future.

We take the conservative view that the failure of a processor evaluating a function must result in a system state where it appears that the function evaluation was never begun. This can be accomplished in a straightforward way with transactions, as we shall see in Chapter 3.

Processor failures can only be detected when a failed processor returns to service. Until then, it is impossible to discriminate between a failure in the communication system and a processor failure. It would be possible to assume that a processor had failed if it did not respond within a predefined time, but, as we shall see, there are cases when we will attempt to use unavailable processors as a matter of course in computations that ultimately succeed.

2.4.2 Algorithm

rp | Eval is described in two stages. First, we show how to implement Weak-Remote-Eval, a weaker form of remote evaluation. Weak-Remote-Eval has miserable semantics, but it is a useful device that allows us to quickly dispense with the details of messages. From this base it will be a simple matter to construct rp | Eval. Weak-Remote-Eval is defined as follows:

Weak-Remote-Eval[processor: Processor, form: Form / result: Any]

> Weak-Remote-Eval will evaluate *form* at the specified processor, returning the result of the evaluation. The function may be evaluated partially or completely any number of times, and the result returned will be from an arbitrary complete evaluation. Furthermore, *form* may be unpredictably evaluated again at indefinite times in the future after Weak-Remote-Eval returns. However, Weak-Remote-Eval does ensure that two evaluations of *form* do not occur at the same time.
>
> If the evaluation of *form* results in Error['Discard] then the remote processor will abandon the evaluation and not produce a response.

With Weak-Remote-Eval we can create *global functions.* A global function is a function that can be passed between processors. Global functions are only useful when an application can tolerate the semantics of Weak-Remote-Eval. We will use global functions in Chapter 3. The following function will create a global function:

Create-Global-Function[fn-name: Atom / gf: Global-Function]

> Create-Global-Function creates a global function. Apply[gf, List[x]] will result in Apply[fn-name, x] being evaluated on the processor where *gf* was created.

```
Create-Global-Function[fn-name: Atom / gf: Global-Function] ← Prog[ [];
        gf ← List['Lambda,
                List['x],
                List['Weak-Remote-Eval,
                        List['Quote GetMyProcessor[]],
                        List['Quote, List['Apply, List['Quote, fn-name], 'x]]
        ]];
        Return[gf];
        ];
```

The implementation of Weak-Remote-Eval is straightforward. At the originating processor a request message containing the form to be evaluated is fabricated and sent at regular intervals to the remote processor until a matching result message is received. At the remote processor request messages are received, and if the processor is not already working on an incoming request, a new process is created to process it. Remote evaluations are assigned unique identifiers so a remote

processor can identify request retransmissions, and so responses can be matched to requests.

Before a form is sent to a remote processor all of the local variables in the form must be evaluated. The function Pre-Eval is defined to do this.

```
Pre-Eval[form: Form / result: Form]
```

Pre-Eval evaluates the atomic arguments in *form* and quotes them. Eval[Pre-Eval[x]] is equivalent to Eval[x].

```
Pre-Eval[form / result] ← Prog[ [];
        IF Null[form] THEN Return[NIL];
        IF Atom[form] THEN Return[List['Quote Eval[form]]];
        IF Eq[car[form], 'Quote] THEN Return[form];
        Return[Cons[car[form], Pre-Evlis[cdr[form]]]];
        ];

Pre-Evlis[x: List / result: List] ← Prog[ [];
        IF Null[x] THEN Return[NIL];
        Return[Cons[Pre-Eval[car[form]], Pre-Evlis[cdr[form]]]];
        ];
```

The model implementation of Weak-Remote-Eval follows.

```
Message ← Record [
        destination: Processor,
        source: Processor,
        id: UniqueID,
        -- reply is T if form is a result
        reply: BOOLEAN,
        form: Any
        ];

Request: Type ← Record [
        message: Message,
        done: BOOLEAN,
        cv: ConditionVariable
        ];

Incoming: Set ← Set-Create[];
Outgoing: Set ← Set-Create[];

Weak-Remote-Eval[processor: Processor, form: Form / result: Any] ← Prog [
        [request: Request; message: Message];
        request ← Create[Request];
        message ← Create[Message];
        -- create message to remote processor
        message.destination ← processor;
        message.source ← GetMyProcessor[];
        message.id ← GetUniqueID[];
        message.reply ← NIL;
```

```
-- evaluate local variables
message.form ← Pre-Eval[form];
request.message ← message;
request.cv ← CreateConditionVariable[RetransmitTime];
request.done ← NIL;
-- note that we have made request
Set-Insert[Outgoing, message.id, request];
UNTIL request.done DO [
        -- send request
        Send[destination, Encode[message]];
        -- wait for response
        Wait[request.cv];
        ];
-- we received our response
Set-Delete[Outgoing, message.id];
Return[request.message.form];
];
```

Receiver, ProcessReply, ProcessRequest, and Request-Eval comprise the rest of Weak-Remote-Eval. Receiver listens for messages, and runs in every processor that services requests. ProcessReply handles results from earlier request messages. ProcessRequest acts on a request to apply a function, forking off Request-Eval to process the evaluation if the request is not already being processed.

```
Fork['Receiver[]];

Receiver[] ← Prog [ [m: Message];
        DO [
                m ← Decode[Receive[]];
                IF m.reply THEN ProcessReply[m]
                ELSE ProcessRequest[m];
                ];
        ];

ProcessReply[m: Message] ← Prog [ [request: Request];
        -- Reply. Notify waiting process.
        request ← Set-Lookup[Outgoing, m.id];
        IF Not[Or[Null[request], request.done]] THEN [
                request.message ← m;
                request.done ← T;
                Notify[request.cv];
                ];
        ];
```

```
ProcessRequest[m: Message] ← Prog [ [old: Message];
        -- if we are already working on this evaluation, don't start
        -- a new worker process.
        old ← Set-Lookup[Incoming, m.id];
        IF Null[old] THEN [
                Set-Insert[Incoming, m.id, m];
                Fork['Request-Eval[m]];
                ];
        ];

Request-Eval[m: Message] ← Prog [ [];
        m.destination ← m.source;
        m.source ← GetMyProcessor[];
        m.reply ← T;
        m.form ← Eval[m.form];
        IF m.form#Error['Discard] THEN Send[m.destination, Encode[m]];
        Set-Delete[Incoming, m.id];
        ];
```

Using Weak-Remote-Eval we can now proceed to describe the algorithm for rp | Eval.

rp | Eval uses connections to implement exactly once semantics and processor failure detection. A connection associates a unique identifier and a sequence number with every request. As shown below, Remember-Eval uses a connection's unique identifier and sequence number to ensure that a form is evaluated exactly once. Connections are stored in a volatile set. Thus, if a processor fails it will forget all of its connections. Remember-Eval always responds to unknown connection identifiers with Error['ProcessorFailure], which implements processor failure detection.

A connection's identifier is generated by the remote processor to ensure that the following sequence of events after a connection is closed is harmless: (1) a delayed copy of the open connection request arrives, (2) a delayed copy of the first request referenced to the connection arrives. If this happens a connection will be established, but the delayed request will be ignored because its unique identifier will be incorrect.

When a request is received at a remote processor, one of the following statements will be true:

1. The request has not been received before, and it is the next request that should be processed. In this case, its sequence number will be one larger than the sequence number of the last request, and it will not appear in the set HeldResults. The request is evaluated, its result is placed in HeldResults (indexed by the request's identifier and sequence number), and the result is returned to the originating processor.

2. The request has been previously received and processed, but the communication system lost the reply message to the originating processor. In this case, the result of the request is in the set HeldResults. The result is retrieved from HeldResults and returned to the originating processor.

3. The request is a delayed duplicate, and the originating processor has already received the results of the request. In this case, the sequence number of the request will be obsolete, and the results of the request will not be in HeldResults. The request is ignored.

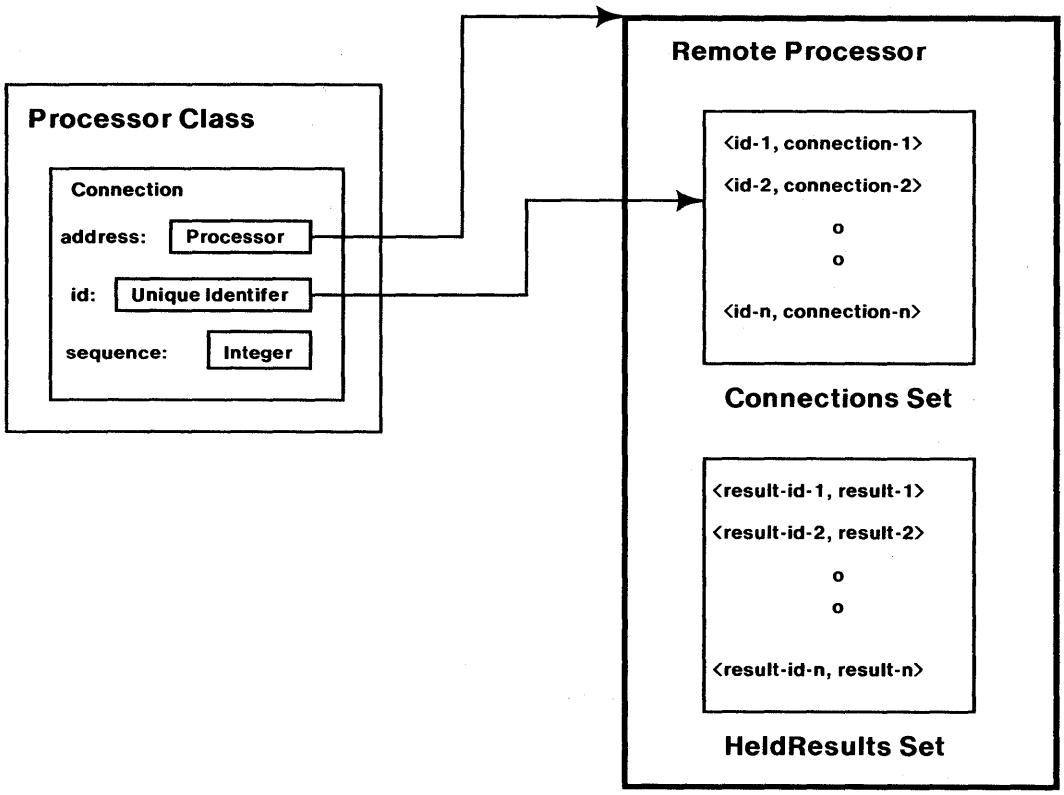
4. The request is a future request that should not be processed yet. In this case, the request will have a sequence number greater than the one expected. The request is ignored. Weak-Remote-Eval will send the request again.

Figure 2.1 shows the structure of a class that is created by Open-Processor. It is possible that more than one connection may be opened by Open-Processor, and that results can be left in HeldResults if an originating processor fails. In a practical implementation time-outs would be used to eliminate these problems.

The following functions constitute the originating processor's part of rp | Eval.

```
Connection: Type ← Record [
        address: Processor,
        id: UniqueID,
        sequence: Integer
        ];

Open-Processor[ref: Processor / rp: Processor-Class] ← Prog [
        [c: Connection];
        -- open connection at remote processor
        c ← Weak-Remote-Eval[ref, 'OpenConnection[]];
        c.sequence ← 0;
        rp ← Create-Class[List[
                'Eval, 'Remote-Eval,
                'Close, 'Processor-Close,
                -- CopyReference is described in Chapter 3
                'CopyReference, 'Default-Copy], NIL];
        -- Instance variables: c
        Return[rp];
        ];

Remote-Eval[form: Form / result: Any] ← Prog[ [next: Connection];
        -- take a ticket (get a sequence number) and increment
        -- the connection's sequence number
        next ← Get-Next-Sequence[c];
        result ← Weak-Remote-Eval[c.address, 'Remember-Eval[next, form]];
        Weak-Remote-Eval[c.address, 'DeleteResult[next]];
        Return[result];
        ];

Processor-Close[] ← Prog [ [];
        -- close the connection at the remote processor
        Weak-Remote-Eval[c.address, 'CloseConnection[c]];
        ];
```

Because many remote evaluations can be serviced by a single processor class at once, care must be taken to use critical sections when assigning and checking sequence numbers.



Structure of a Processor Class

Figure 2.1

```
SL: Lock ← CreateLock[];

Get-Next-Sequence[c: Connection / next: Connection] ← Critical[SL, 'Prog[ [];
        -- increment connection sequence number
        c.sequence ← c.sequence + 1;
        Return[c];
        ]];

Bad-Sequence[c1: Connection, c2: Connection / bad: Boolean] ← Critical[SL, 'Prog[ [];
        IF c1.sequence # c2.sequence THEN Return[T];
        Return[NIL];
        ]];
```

The following functions are executed by remote processors in support of rp | Eval.

```
Connections: Set ← Create-Set[];

OpenConnection[/ c: Connection] ← Prog [ [];
        -- create a new connection
        c ← Create[Connection];
        c.sequence ← 1;
        c.id ← GetUniqueID[];
        c.address ← GetMyProcessor[];
        Set-Insert[Connections, c.id, c];
        Return[c];
        ];

HeldResults: Set ← Create-Set[];

Result ← Record[value: Any];

Remember-Eval[c: Connection, form: Form / result: Any] ← Prog [
        [connection: Connection;  rr: Result];
        rr ← Set-Lookup[HeldResults, c];
        -- if we still have result from previous Eval, return it
        IF Not[Null[rr]] THEN Return[rr.value];
        connection ← Set-Lookup[Connections, c.id];
        -- if connection unknown, assume that we have crashed
        IF Null[connection] THEN Return[Error['ProcessorFailure]];
        -- make sure this is the next in the sequence
        IF Bad-Sequence[connection, c] THEN Return[Error['Discard]];
        -- create a record so we can distinguish the following cases:
        --  a) a result of NIL
        --  b) a result that is not in HeldResults
        rr ← Create[Result];
        -- evaluate the form
        rr.value ← Eval[form];
        Set-Insert[HeldResults, c, rr];
        Get-Next-Sequence[connection];
```

```
        Return[rr.value];
        ];
DeleteResult[c: Connection] ← Prog [ [];
        -- originating processor has received result
        Set-Delete[HeldResults, c];
        ];
CloseConnection[c: Connection] ← Prog [ [];
        -- originating processor closed connection
        Set-Delete[Connections, c.id];
        ];
```

2.5 Summary

This chapter introduced and characterized three types of hardware components: processors, communication, and storage devices. Stable storage was introduced, and identifiers that are unique over processors and time were demonstrated. It was shown how to perform remote form evaluation exactly once with processor failure detection.

Exercises

1. Assume that you have a remote processor class, *rp*. Give a statement that will determine the value of x in the environment at *rp*.

2. Assume that your processor provides you with a stream of uncorrelated but biased bits. Give an algorithm that will remove the bias (hint: the algorithm will not produce unbiased bits at a fixed rate) [von Neumann 51].

3. Give an algorithm for implementing stable storage. Assume that the basic unit of storage is a page, failures are independent, and the probability that a storage device remembers a page after time t is $pr(t) = e^{-\lambda t}$. What is pr(t) for your algorithm? How could you do better?

Chapter 3: Transactional Storage

This chapter presents a basic set of facilities for shared information storage in three stages. The first section of the chapter introduces transactional storage, the second section discusses algorithms for implementing such storage, and the final section outlines some refinements. The information storage system described in this chapter would be ideal if storage system objects never moved, storage did not need its properties improved, there was no need to reconfigure storage, and people were perfectly trustworthy. Each of these problems is the subject of a subsequent chapter.

3.1 Model

Users do not want the inconsistent intermediate results they store to be misinterpreted by others, and thus inherent with the concurrent sharing of storage is the need for coordination. To this end we extend the notion of stable storage to create what we will call *transactional storage.* Transactional storage solves the problem of coordination by providing a client with the illusion that there is no other activity in the system. This illusion is achieved by the use of transactions, which are a basic unit of concurrency control and error recovery.

The following sections define a transactional storage system in terms of an an ideal set of generic objects and operations. We will call objects that store information *files,* and objects that store files *volumes.* Volumes are intended to model storage devices. Not all objects have to support all operations, and it is possible to create specialized types of files and volumes. For example, an indexed file could be defined that implemented storage and retrieval by key. Extensions to the basic facilities are provided in subsequent chapters, and we expect that clients will also create facilities tailored to their needs.

A further property of our transactional storage system is that it hides the physical location of a resource from its clients. Clients will typically use objects without knowing where they are.

The facilities of a transactional storage system are introduced in the following five sections. In order, the topics covered are naming, transactions, volumes, files, and immutable objects.

3.1.1 References

All objects we define are named by *references.* A reference contains a unique identifier that unambiguously identifies its referent. References can be extended to contain such things as location hints, cryptographic keys, indirect pointers, and other useful things. We have already introduced processor references in Chapter 2 with the type *Processor.*

```
Reference ← Record[id: UniqueID];
```

Using references, clients can implement naming systems tailored to their users' needs. A general model for a naming system is a function that maps context dependent specifications into references. For example, a data base could be employed to resolve queries into references. A query might be "show me the file I created last Tuesday". Alternatively, objects could be selected from a menu of icons on a terminal screen. In such a system screen coordinates would be resolved to references.

Records are used to construct references. As described in Appendix A, record instances can be converted to and from arrays of bytes by the operations Encode and Decode. Thus references can be saved in files. Clients can not predict what fields will be in a reference, and thus should store references verbatim. As a matter of convention, the type of a reference will correspond to the type of its referent. Thus, the function Is (Appendix A) can be used to determine the type of object that a reference names.

Every object has a reference-count, which corresponds to the number of *counted* references to the object. An object can also have *uncounted* references, which do not affect its reference-count. The storage for an object is not released until all of its counted references are destroyed. When the storage for an object is released we say the object is *deleted.* If a reference is counted, care must be taken to ensure that it is not lost, and that it is properly destroyed when it is no longer needed. Clients, of course, are not bound to use the full generality of the reference-counting mechanism. They could limit the number of counted references to an object to one, in which case destroying a counted reference would function like delete-object. Counted references include the *Counted* type:

Counted ← Record[];

Reference-counts are provided as a convenience to help with, but not completely solve, the problem of reclaiming objects that are no longer wanted. Reference-counts have traditionally been somewhat suspect, but there is a good chance that they will be more reliable when they are maintained in transactional storage instead of volatile storage.

Garbage collection can also be used to reclaim unwanted objects. An asynchronous garbage collection scheme has been successfully used in a system that keeps systematic track of all counted references [Garnett and Needham 80]. However, as the size of a system grows, it is probably not wise to assume that garbage collection will always be a feasible alternative. Furthermore, when protection is added to a system, a garbage collector may not be authorized to enumerate all of the references in the system.

There is a well known accounting problem with reference-counts. An object may exist for an undefined time after the user that is paying for its storage is interested in it. Provisions could be made to allow users that fund objects to delete them, but the semantics of reference-counts would be destroyed.

It is possible that a configuration will have enough storage capacity that an object will never have to be deleted. As we shall see in Chapter 8, in this case reference-counts are still useful to automatically reclaim unused copies of objects that are stored on the configuration's limited amount of high performance storage.

If a read-only reference is used to refer to an object, no operations can be issued that would change the object in any way. Read-only references provide a client with a form of self-protection. Read-only references include the type *Read-Only.* Add-Type (Appendix A) can be used to add the type Read-Only to a reference.

Read-Only ← Record[];

All of the operations on objects in the system are implemented by classes. To acquire a class that will service operations for a given object, a reference for the object is opened with the function Open. Open tests the type of its input argument and then evaluates a function that is specialized to

deal with the presented type of reference. For example, if Open is applied to a processor reference, Open-Processor will be evaluated. Appendix C contains a complete description of Open. The following operations are supported by all objects.

Open[ref: Reference, tc: TC, ring: List[Key], guards: List[Key] / c: Class]

> Open returns a class that will service requests to *ref*. All operations that the class performs will be part of transaction class *tc*, as described in Section 3.1.2. The precise initialization steps that Open takes depend on the reference's type. If a reference's referent can not be found, Error['NotFound] is returned. The third and fourth arguments to Open, *ring* and *guards*, are described in the chapter on protection (Chapter 7).

c: Class | Close[]

> Close deactivates a class. Open references should be closed after they are no longer being used.

c: Class | GetID[/ id: UniqueID]

> GetID returns the unique identifier of the object serviced by *c*.

c: Class | GetTransactionClass[/ tc: TC]

> GetTransactionClass returns the transaction class of class *c*.

c: Class | CopyReference[counted: Boolean / ref: Reference]

> An open reference can be copied, and a client can specify whether it wishes the copy to be counted or uncounted. Making a counted reference is the only way to increment the reference-count of an object.

c: Class | DestroyReference[/ deleted: Boolean]

> When a reference is no longer needed, it should be destroyed. The only way to decrement the reference-count of an object is to destroy one of its counted references. DestroyReference returns T if the object's reference-count went to zero. If the reference count of the object went to zero, the object is deleted when *c* is closed.

3.1.2 Transactions

Function evaluations that access transactional storage, also called *actions*, are grouped into disjoint sets called *transactions*. A transaction is defined to have three properties.

> *Totality.* Totality ensures that all of the actions in a transaction will either occur exactly once or not at all. Because transactional storage is built from stable storage, totality also guarantees that if a transaction appears to occur its effects will be resilient to hardware failures.
>
> *Serial Consistency.* Serial consistency makes it appear to a transaction that that there is no other simultaneous activity in transactional storage.
>
> To formalize the notion of serial consistency we will adapt some ideas from [Eswaran et al. 76]. As we have stated, a transaction is a set of actions. For simplicity, we will only

consider read and write actions, and we will assume that a transaction performs at most one read and one write action on a data element.

The question of serial consistency arises when a transactional storage system concurrently executes actions drawn from several transactions. Imagine two simple transactions:

T1	T2
(a11) Read Y	(a21) Read X
(a12) Write X	(a22) Read Y
(a13) Write Y	(a23) Write X

The order in which the actions of T1 and T2 are processed is called a *schedule.* A schedule is an arbitrary interleaving of the actions of a set of transactions into a single sequence. A *serial schedule* results when transactions are executed one at a time to completion. Thus, there are two possible serial schedules for our example:

Sa: {a11, a12, a13, a21, a22, a23}
Sb: {a21, a22, a23, a11, a12, a13}.

A schedule generates a *dependency* relation. A dependency relation describes transactions that depend on one another. If S is a schedule, then $\langle T_a, e, T_b \rangle$ is a member of DEP(S) if:

T_a and T_b are distinct transactions
$a_i \in T_a$ and $a_j \in T_b$
a_i and a_j are actions from S
a_i occurs before a_j in S
a_i is a write action to a data element *e*
there are no other write actions to *e* between a_i and a_j
a_j is a read or write action to *e.*

In other words, $\langle T_a, e, T_b \rangle$ is in DEP(S) if T_a updates data element *e* that T_b uses.

If two schedules S and S* have identical dependency relations, DEP(S) = DEP(S*), then they provide each transaction with the same inputs and outputs. Thus, two distinct schedules that have identical dependency relations are said to be *equivalent.*

A system provides serial consistency if it guarantees that the schedule it will use to process a set of transactions is equivalent to one of the possible serial schedules.

External Consistency. External consistency guarantees that a transaction will always receive current information. Using the concepts we have just introduced, we can provide a formal definition of external consistency. The actual time order in which transactions complete defines a unique serial schedule. This serial schedule is called the *external schedule.* A system is said to provide external consistency if it guarantees that the schedule it will use to process a set of transactions is equivalent to its external schedule.

Let's consider a classical example to see why these properties are important in practice. Imagine that a bank teller requests that the money in a customer's savings account be transferred to his checking account. The bank's computer system would begin a transaction, and evaluate functions referenced to this transaction to withdraw the balance of the customer's savings account

and deposit it in his checking account. Totality ensures that either both the withdrawal and deposit will take place, or neither of them will. Serial consistency eliminates problems that might result from concurrency. For example, it prevents two independent transactions from observing the same balance, and then simultaneously withdrawing it. Finally, external consistency ensures that if the customer goes to another teller to cash a check, the teller will see the customer's new checking account balance.

A transaction is *active* until it ends by either *committing* or *aborting*. If it commits, the effects of its actions will become permanent and public. If it aborts, the effects of its actions will be undone. If a client does not wait for an action to complete before requesting its transaction to commit, the action may or may not occur. Furthermore, a transaction can commit with outstanding uncompleted reads. Normally a client controls the destiny of a transaction, but transactions can be spontaneously aborted by the system to cope with exceptional conditions such as hardware failure or the inability to guarantee consistency.

In addition to performing actions in transactional storage, a transaction can cause *real actions* to occur [Gray 78]. A real action is an action that once done, can not be undone. Examples of real actions are launching a missile or dispensing cash from an automated teller. Totality is guaranteed for real actions as well as for updates to transactional storage. For example, assume a withdrawal transaction from an automated teller updates transactional storage (to debit the customer's account) and dispenses money from the teller machine. Totality guarantees that either both of these actions will occur, or neither of them will.

Committing a transaction is a real action. The only way to undo the effects of a comitted transaction is to execute a *compensating* transaction. If compensation is possible, precisely how it can be accomplished is highly application dependant. For example, it is very difficult to recall a missile that has been launched.

A benefit of the all-or-nothing nature of transactions is that they can be used to insulate remote evaluations from processor failures. We assume that every evaluation of a transactional storage system function is associated with a transaction. If a processor fails while it is executing such an evaluation, it is sufficient to abort the evaluation's transaction. All of the effects of the evaluation will be discarded. Furthermore, if orphans later try to execute actions referenced to the aborted transaction, they will not do any damage.

A *coordinator* is an object that creates and manages transactions. When a coordinator is opened, the transaction parameter to Open is ignored. The interface to coordinators is as follows:

```
Transaction ← Extend[Reference, Record[]];
Coordinator ← Extend[Reference, Record[]];

cc: Coordinator-Class | Create-Transaction[/ t: Transaction]
```

> Create-Transaction creates a new transaction and returns a reference for it. The transaction reference returned is not counted. However, if a counted reference for the transaction is made, the transaction will persist after it is committed or aborted. Abort and Commit erase transactions that have a zero reference-count.

Once a transaction has been created, it can be opened. The transaction parameter to Open is ignored when a transaction is opened. The type of class that results from opening a transaction will

be often referred to as a *TC*, for transaction class. The operations that a transaction provides are as follows:

tc: Transaction-Class | Commit[]

Commit causes the effects of a transaction to be made permanent and public.

tc: Transaction-Class | Abort[]

Abort erases any effects of a transaction and makes it appear as if the transaction had never existed.

tc: Transaction-Class | GetStatus[/ status: Atom]

GetStatus returns C, A, or NIL, for a transaction that is committed, aborted, or active, respectively.

Clients that hold information derived from an active transaction are called *participants*. A buffer manager is an example of a participant. At commit, a participant should ensure that all updated information has been output to transactional storage. At commit or abort, a participant may wish to invalidate information that was derived from the transaction. Because the transaction is no longer active, the derived information is no longer guaranteed to be current.

tc: Transaction-Class | AddParticipant[commit, abort: Global-Function / id: UniqueID]

A client can be informed when a transaction commits or aborts by calling AddParticipant. The functions provided by the client will be applied to the specified transaction's reference just before the transaction commits or aborts, respectively.

The final disposition of a transaction is not guaranteed when *commit* or *abort* are called. Thus, the commit function can not be used to indicate success to a user. Only a successful return from tc | Commit indicates that a transaction has sucessfully committed.

tc: Transaction-Class | DeleteParticipant[id: UniqueID]

DeleteParticipant deletes a participant. The participant to be deleted is specified by a unique identifier that was returned from AddParticipant.

It is possible that a client will request a transaction to commit and then its processor will fail before it can find out what happened. A solution to this problem adopted by some systems is to use real actions to indicate success. Alternatively, a client can keep a counted reference for the transaction, and use GetStatus to discover what happened when its processor recovers.

3.1.3 Volumes

A volume is a file-storage service that is independent of a specific hardware realization. A volume can be implemented by a fraction of a storage device, one storage device, or more than one storage device. The details of how a physical medium are prepared for use are beyond the scope of this work, and we simply assume that a mechanism exists that will create a new volume and return a reference for it.

The storage technology used to implement a volume influences its properties. We call the volumes implemented by serial devices and write-once devices *serial volumes* and *write-once*

volumes, respectively. A volume may be both serial and write-once. Serial and write-once volumes influence the properties of the files stored on them, as described in the next section.

A volume is named by a volume reference, and serial and write-once volumes are further distinguished by unique types of volume references. The types that are used to implement these references are:

```
Volume ← Extend[Reference, Record[]];
Serial ← Record[];
Serial-Volume ← Extend[Volume, Serial];
Write-Once ← Record[];
Write-Once-Volume ← Extend[Volume, Write-Once];
```

Users may choose not to rely on reference-counts to determine when a volume should be retired, and thus an implementation of volumes may choose not to implement the reference-counting semantics of CopyReference and DestroyReference. However, automatic volume reclamation may be appropriate for some storage devices [IBM 80b].

Create-File is the only operation that all volumes must support. However, a volume might choose to implement an operation to return its remaining storage capacity, or an operation to enumerate the files that it holds.

3.1.4 Files

A file is a variable-length array of bytes that can be read and written. The length of a file can be set, and the system may or may not remember information past the declared end of a file. A file is initialized to contain all zero bytes, and an attempt to read past the end of file will return zero bytes. If data is written beyond the end of a file, the file will be automatically extended and have its length adjusted.

Like other objects, a file is named with a reference. A file reference contains a reference for its containing volume. A file reference may also contain other things, such as storage device addresses, but as we described in the section on references, the job of a client is simply to store references verbatim. A file reference is defined as:

```
File ← Extend[Reference, Record[volume: Volume]];
```

Files that are stored on serial and write-once volumes have special properties and reference types. If a file is stored on a serial volume, then non-serial access to the file is very expensive. If a file is stored on a write-once volume, then it can only be written once. It is unlikely that clients will have to deal directly with serial or write-once files. In Chapter 8 we discuss briefly how to obtain the cost and performance benefits of serial and write-once devices while presenting a normal file interface. Serial and write-once file references are defined as:

```
Serial-File ← Extend[File, Serial];
Write-Once-File ← Extend[File, Write-Once];
```

The following functions define the file operations. Files can be created, read, written, and they can have their length changed.

vc: Volume-Class | Create-File[name: UniqueID, exists: Boolean / ref: File]

Create-File creates a file with a given name on a specific volume, and returns an uncounted reference for the file. If a client wishes to preserve the file, it must make a counted reference for it. If a file already exists on the specified volume with the given name, Error['DuplicateIdentifier] is returned unless exists is T. Although Create-File is described in this section, it is intrinsically a volume operation.

fc: File-Class | Write[startPage: Integer, pages: Integer, data: Byte-Array]
fc: File-Class | Read[startPage: Integer, pages: Integer / data: Byte-Array]
fc: File-Class | Set-Size[pageCount: Integer]
fc: File-Class | Get-Size[/ pageCount: Integer]

The fundamental operations for data retrieval and storage are read and write. Files are read and written in units of pages, and the size of a page is implementation dependent. Write writes into a file from an array a specified number of pages starting at a given page number. Read reads from a file into an array a specified number of pages starting at a given page number. The length of a file can be set and determined by Set-Size and Get-Size, respectively.

These operations can result in four exceptional conditions. An attempt to write or set the length of a file on a read-only device returns Error['ReadOnly]. An attempt to write or set the length of an immutable file (see below) returns Error['Immutable]. An attempt to extend the length of a file, either by writing past the end of it or by an evaluation of Set-Size, will return Error['InsufficientCapacity] if the file's containing volume is full. If an unrecoverable failure is detected Error['StorageDamaged] is returned.

3.1.5 Mutable and Immutable Objects

After an idea originally suggested by Paul McJones, files and volumes can be freely modified until they are made *immutable*. Once an object is made immutable it becomes permanently read-only, and there is no way of making the object mutable again. An immutable file can not be written or have its length changed, and an immutable volume consists of a permanently fixed set of immutable files.

The following two functions set and test the immutable property:

or: Class | SetImmutable[]
or: Class | IsImmutable[/ immutable: Boolean]

Immutable objects afford important practical benefits. A copy of an immutable object can be made without fear that the copy will become obsolete. Furthermore, because immutable objects can not change, the concurrency control and error recovery facilities of transactions are not required for their contents. However, deleting an immutable object is a real action, and thus requires the facilities of transactions.

Theoretically a storage system could consist of a large number of immutable files and one mutable file that contained a single reference. In such a scheme, every time an object was updated, a new immutable version of the storage system would be created. The single mutable reference would point to the current version of the storage system. The extent to which immutable objects

should be used will depend on the characteristics of an application under consideration, and the relative cost of mutable files.

The type of device that is used to store an object and whether the object is immutable or not are completely independent. For example, objects that are stored on a write-once storage device do not have to be immutable. Although an object that is stored on a write-once device can not be directly updated, it could be moved to a read-write device, and then updated.

3.2 Basic Algorithms

Techniques for implementing single-processor, transactional-storage systems have been well understood for some time. At the time of this writing, a few systems have been sucessfully constructed that implement a transactional storage system across processor boundaries [Eade et al. 77; Israel et al. 1978; LeLann 81; Tandem 81]. Because we are drawing on previous work, we will provide an overall sketch of how a decentralized transactional storage system can be implemented, but the reader should consult the references provided for details.

Our survey is given in four sections. Transactional storage is first introduced on a single processor, and then extended to the decentralized case. We then discuss a different algorithm for transactional storage due to Reed. The final section describes how remote objects are accessed in our model.

3.2.1 Single Processor Case

The job of a transactional storage system is to implement the three properties of transactions: totality, serial consistency, and external consistency. Existing systems follow a common pattern [Gray 78].

The two types of consistency are implemented by lock management. A lock protects objects that a transaction uses from concurrent updates from other transactions, and it notifies other transactions that an object the transaction updates is busy. Locks are typically set implicitly as the result of storage system function evaluations. A client can also use its knowledge of what it is trying to accomplish to directly set and clear locks as an optimization. Some form of time out mechanism is usually provided that will cause a transaction to be aborted if it holds a popular lock too long.

Totality is implemented by recovery management. In an "undo" implementation, updates are directly applied to objects, and recovery management remembers old values in case a transaction aborts. If a transaction aborts, recovery management undoes the transaction's updates by consulting the preserved old values; hence the designation undo. In a "redo" or "intentions" implementation a transaction does not directly update objects. Requested updates are remembered by recovery management, and if a transaction commits, recovery management applies them to their respective objects. Thus, the list of updates remembered by recovery management are called *intentions.* If the system fails while intentions are being applied they must be reapplied, or redone, until they have been completely applied.

Recovery management requires storage. In an undo scheme the extra storage is for old values, and in a redo scheme the extra storage is for new values. This storage is implemented in existing systems with a *log* or a *shadow mechanism.*

A log is an append-only file that records the history of the system. In typical log-based systems, updating an object causes a log record to be written that includes the object's name, old

and new values, and information that identifies the update's transaction.

A shadow mechanism implements storage for new values. New values are written out to fresh pages called shadows. If a transaction commits, any files that were updated have their physical storage description changed to point to their shadow pages instead of to pages that contain old values.

The ideas of undo and redo can be combined. For example, in System R performance is increased by not keeping stable storage up to date. If the system fails, updates may have to be undone or redone from the log [Gray et al. 79].

Because most log-based systems remember redo values, they implement stable storage. Even if part of a file is lost, the log will still have a copy of the data. To prevent logs from becoming infinite, condensed snapshots called fuzzy dumps are taken periodically [Gray et al. 79]. Systems that only use shadows must implement stable storage with an additional mechanism.

Recovery management keeps track of the real actions that a transaction has requested. These actions are postponed until recovery management is told to commit or abort. If recovery management is told to commit a transaction, then the transaction's real actions are performed.

Now that we have introduced the concepts of lock and recovery management we can describe how they are synchronized to implement transactions. Our discussion assumes that a transaction is processed by more than one lock manager and more than one recovery manager. Inside of a single system there may be several lock and recovery managers. This commonly occurs when two independent data base systems are integrated.

As we have discussed, a transaction is aborted for one of two reasons. First, a client can request that a transaction be aborted. Second, if a lock manager can not guarantee the consistency of a transaction, the transaction is aborted. When a transaction is aborted, the transaction's coordinator tells the transaction's lock managers to release the transaction's locks, and the transaction's recovery managers to erase the transaction's updates.

A transaction is committed only if a client requests that it be committed. When a client requests a transaction to commit, the transaction's coordinator executes the following two step algorithm.

1. The coordinator asks all of the transaction's lock managers to guarantee the transaction's consistency, and all of the transaction's recovery managers to guarantee to commit on demand. If any manager can not make the requested guarantee, the transaction is aborted.
2. After all of the managers have made their promises, the coordinator informs the recovery managers to go ahead and commit the transaction. After the recovery managers have applied the transaction's updates, the lock mangers are told to release the transaction's locks.

This two step process is known as the two-phase commit protocol.

3.2.2 Decentralized Case

The decentralized case is not substantially different from the single processor case with multiple lock and recovery managers. A transaction that spans processors is implemented by a cooperating set of single processor transaction systems. Each processor, with its own transactional storage system, is called a *worker*. A worker implements lock and recovery management for the data it stores.

A decentralized transaction's coordinator carefully synchronizes the commit and abort processing of the transaction's workers as follows. If a transaction is aborted, either because a lock manager could not guarantee consistency or because the client requested an abort, the coordinator tells every worker to abort. If a client requests commit, the coordinator asks every worker to promise to commit on demand. Before a worker can make such a promise it must ensure that as far as it knows the transaction is consistent, and it has all of the information in stable storage that is required for the transaction to commit. If these conditions are met then the worker can make its solemn promise. A worker that has made such a promise is said to be prepared. After making its promise a worker must remain prepared until it is informed to commit or abort by the coordinator. The request and promise sequence is phase one. If all workers promise to commit, then they can all be instructed to go ahead and commit. This is phase two. If any worker fails or refuses to make the promise then the coordinator aborts the transaction. The two steps of a decentralized commit, request and receive promises, followed by commit, is the decentralized version of the two-phase commit protocol [Gray 78].

Coordinators must be very reliable so workers are not left prepared for an appreciable length of time. A prepared worker is undesirable because its lock manager can not permit any transactions to be processed that would destroy the consistency of its prepared transaction. If a transaction's coordinator fails a worker may remain prepared for a long time.

We now digress for a moment to introduce a necessary idea. Imagine that a value can be replicated at a number of distinct places and manipulated in a way that is reasonably tolerant of place failures. There are two operations to manipulate the value: TryToSet and ReadValue. TryToSet attempts to set the value, but it only sets the value if it was NIL. Thus, if many TryToSet operations are simultaneously evaluated only one of them will successfully update the value. TryToSet will not return until the value is not NIL (it may or may not have succeeded). ReadValue returns the current value. Several algorithms have been proposed that solve this problem [Lamport 78a; Lampson 80; Thomas 79].

The following addition to the synchronization algorithm sketched above keeps workers from depending on a coordinator. The coordinator stores the state of a transaction at a number of places using TryToSet. The state of a transaction can be NIL, C, or A, representing active, committed, or aborted. When a transaction is aborted, the coordinator executes TryToSet['A]. If a client requests that a transaction be commited, and the coordinator has received promises from every worker, then the coordinator executes TryToSet['C]. After it has done this, the coordinator sees if the value of ReadValue is C, and if so, it tells the workers to commit. When a worker is asked to get prepared and make its promise, it is passed the list of places where the state of the transaction is stored. If a worker gets restless or its lock manager notes that there are competing requests for a prepared transaction's locks it can use ReadValue to determine the state of the transaction. If the state is C, it can go ahead and commit. If the state is A, it can abort. If the state is NIL, it can execute TryToSet['A]. Thus, regardless of when the coordinator might fail the transaction will continue to move towards termination.

3.2.3 Reed's Algorithm

Reed has proposed a new way of implementing transactions [Reed 78]. We will not describe his scheme in detail, but we briefly sketch how his assumptions differ from tradition. First, he

discarded the idea of private storage for recovery management, and instead let new and old values coexist in storage. New values are hidden by a version mechanism until a transaction commits, and old values persist for an implementation dependent period. Second, he abandoned locks as a concurrency control mechanism. Every transaction in his system is assigned a distinct pseudo-time period during which it appears to run. Data items have time stamps acquired from transactions, and rules are enforced that ensure serial consistency.

3.2.4 Located References

Open is defined to return a class that will service requests for an object regardless of where the object is located. In this section we will make simplifying assumptions so we can clearly present the mechanics of how remote objects are accessed. In later chapters we add more complex concepts, such as object location.

A reference is said to be located if it includes the address of a processor that will service requests for its referent. Until Chapter 4 we assume that all references are located, and that the locations in references are always accurate. The address of an object is added to its reference as follows:

Create-Located[ref: Reference, loc: Processor / lref: Located]

> Create-Located creates a located reference. When Open is applied to *lref*, control will be transferred to processor *loc*, and *ref* will be opened there.

```
Located ← Record[ref: Reference, loc: Processor];

Create-Located[ref: Reference, loc: Processor / lref: Located] ← Prog[ [];
        lref ← Create[Located];
        lref.ref ← ref;      lref.loc ← loc;
        Return[Add-Type[lref, Major-Type[ref]]];
        ];
```

When we create references that are derived from other references, the new reference is assigned the same major type as the reference it was derived from. For example, if Create-Located is applied to a file reference, it will return a located file reference.

```
Major-Type[ref: Reference / type: Type] ← Prog[ [];
        -- return the major type of ref
        Return[Intersection[ref.type,
                Union[File, Volume, Coordinator, Transaction, Index, Processor, Counted,
                Secure-Door, Suite]];
        ];
```

When Open encounters a located reference, it calls Open-Located:

Open-Located[ref: Reference, tc: TC, ring: List[Key], guards: List[Key] / c: Class]

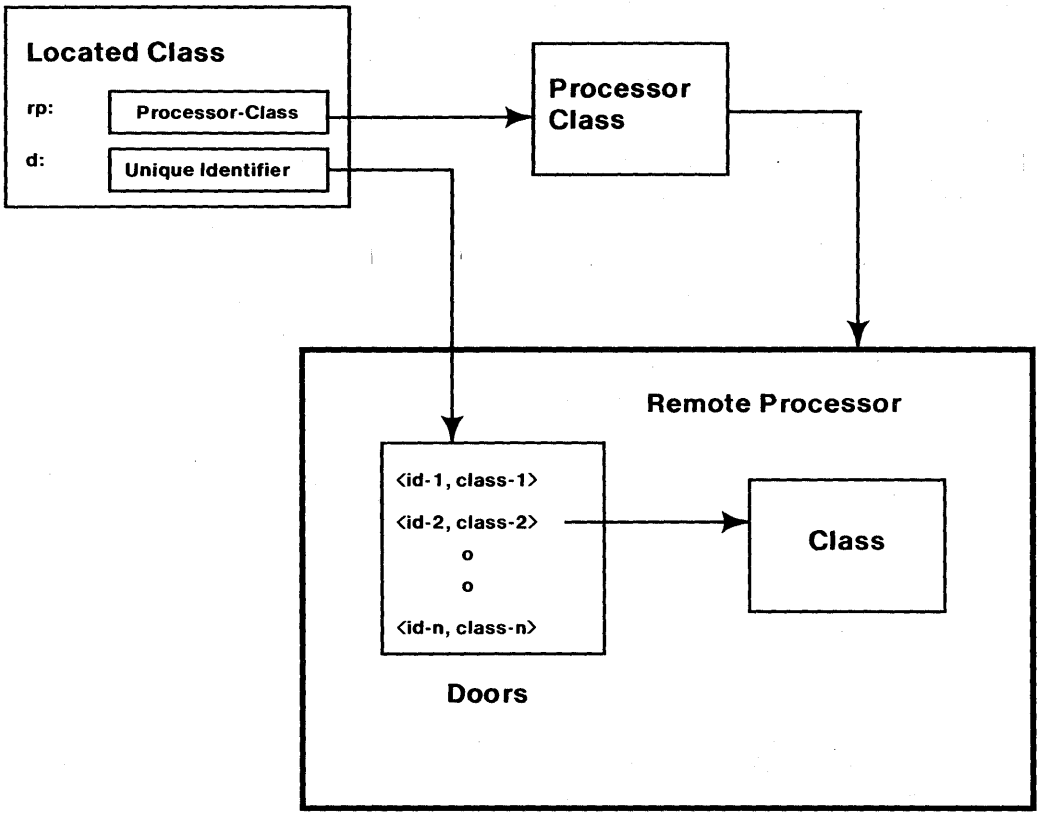
> Open-Located will open *ref* on the processor that is specified in its reference. The class returned can be used to communicate with the remote object as if it were local.

It is not sufficient just to evaluate Open on the remote processor because classes can not be

passed between processors. Thus, Open-Located arranges to open a reference on a remote processor, and have the resulting class kept at the remote processor in a set called *Doors*. An entry in Doors is indexed by a unique identifier, and the unique identifier is returned to the originating processor. We will call an entry in the set Doors a *door*. The class created by Open-Located can refer to a remote class by specifying the unique identifer of its door.

Shown below is the model implementation of Open-Located. Open-Door opens a reference on a remote processor and creates a door for the resulting class. A request directed at a located class is sent to Enter-Door. Enter-Door passes the request and a door identifier to the remote processor, where Door-Eval actually performs the requested operation. Figure 3.1 shows the structure of the class that Open-Located creates.

```
Open-Located[ref: Reference, tc: TC, ring: List[Key], guards: List[Key] / class: Class] ←
Prog [
        [rp: Processor-Class; d: Door; t: Transaction; r: Reference];
        IF ref.loc=GetMyProcessor[] THEN [
                -- object is at this processor
                class ← Open[ref.ref, tc, ring, guards];
                IF Is[class, Error-Type] THEN Return[class];
        ] ELSE [
                -- object is at a remote processor
                t ← tc | CopyReference[];
                r ← ref.ref;
                rp ← Open[ref.loc];
                -- pass guards but not ring to remote processor  (details in Chapter 7)
                d ← rp | Eval['Open-Door[r, t, guards]];
                IF Is[d, Error-Type] THEN Return[d];
                class ← Function[Enter-Door];
                ];
        -- Instance variables: rp, d, ref, tc
        Return[Create-Class[
                List['CopyReference, 'Default-Copy],
                class]];
        ];

Enter-Door[request: List / result: Any] ← Prog [ [];
        result ← rp | Eval['Door-Eval[d, request]];
        -- Eliminate problems with orphans if the processor fails
        IF result=Error['ProcessorFailure] THEN tc | Abort[];
        Return[result];
        ];

Default-Copy[counted: Boolean/ cref: Reference] ← Prog [ [];
        -- default implementation of CopyReference
        cref ← Copy[ref];
        IF counted THEN [
                -- if counted, increase reference count of referent
                Apply[superclass, request];
                -- add counted type
                Add-Type[cref, Counted];
                ]
```



Structure of a Located Class

Figure 3.1

```
        ELSE Remove-Type[cref, Counted];
        Return[cref];
        ];
```

Open-Door opens a reference and returns the identifier of its door. Door-Eval applies a door to a request.

```
Doors ← Create-Set[];

Open-Door[ref: Reference, t: Transaction, guards: List[Key] / d: Door] ← Prog [
        [c: Class;   dc: Class];
        c ← Open[ref, Open[t], NIL, guards];
        IF Is[c, Error-Type] THEN Return[c];
        d ← GetUniqueID[];
        dc ← Create-Class[List['Close, 'Close-Door], c];
        Set-Insert[Doors, d, dc];
        -- Instance variables: c, d
        Return[d];
        ];

Door-Eval[d: Door, request: List / result: Any] ← Prog[ [c: Class];
        c ← Set-Lookup[Doors, d];
        IF Null[c] THEN Return[Error['NoSuchDoor]];
        Return[Apply[c, request]];
        ];

Close-Door[] ← Prog [   [tc: Transaction-Class];
        tc ← c | GetTransactionClass[];
        c | Close[];
        tc | Close[];
        Set-Delete[Doors, d];
        ];
```

3.3 Refinements

3.3.1 Lock Compatibility

A typical locking structure that is used to guarantee serial consistency has two types of locks, read and update. These locks are set on data items implicitly in response to file operations. The compatibility of the locks is specified by Table 3.1.

	No Lock	Read	Update
No Lock	Yes	Yes	Yes
Read	Yes	Yes	No
Update	Yes	No	No

Table 3.1: Typical Lock Compatibility Matrix

A transaction is suspended if it attempts to set a lock that is incompatible. This matrix corresponds to the familiar rule that either *n* readers or one updater are permitted to access a file

simultaneously.

This locking rule potentially can introduce long periods of time when information is unavailable. For example, if a user controls the length of a transaction, he can hold a update lock for a long period of time. This may naturally occur as a user thinks at the keyboard.

A property of serial consistency is that all of a transaction's updates appear to occur at transaction commit time. One can take advantage of this property to increase the concurrency in the following way. Updates appear to occur when they are issued, but in fact are buffered until commit time by the stable storage system. Allowances are made so a read following a write will receive the write's data. When a user updates a datum, an I-Update lock is set, for intention to update. At commit time I-Update locks are converted to Commit locks, and the updates are actually reflected in stable storage. Table 3.2 specifies the new lock compatibility matrix.

	No Lock	Read	I-Update	Commit
No Lock	Yes	Yes	Yes	Yes
Read	Yes	Yes	Yes	No
I-Update	Yes	Yes	No	No
Commit	Yes	No	No	No

Table 3.2: Lock Compatibility Matrix with Intention Locks

With this revised locking matrix, information is only unavailable for predictably short periods of time, during commit processing. This results in increased concurrency, but it may cause the later abortion of a transaction. We chose to make multiple I-Update locks incompatible, because eventually one of the two transactions would probably commit, and become incompatible. Thus we chose not to postpone the inevitable.

This is one example of what is called *lock conversion* [Gray 78]. Lock conversion allows clients to predeclare their intentions to help lock management schedule transactions.

3.3.2 Lock Granularity

The granularity of a lock [Gray et al. 76] refers to its scope. For example, a fine grained lock might be set on a range of bytes in a file, and a coarse grained lock might be set on an entire file. Choosing the granularity of locks presents a trade off between concurrency and lock manager overhead. Fine grained locks increase concurrency by not locking as much, but lock management will set more locks. On the other hand, if coarser locks are chosen, then concurrency will be reduced.

Locking protocols have been devised that provide variable granularity locks [Gray et al. 76]. Variable granularity locks allow clients to decide how much should be locked, and have proven to be very valuable in practice.

3.3.3 Lower Degrees of Consistency

There are a number of different levels of consistency, but serial consistency is the highest, and the consensus is that lower degrees of consistency are not sufficient for all applications [Gray 78]. [Gray et al. 76] define four degrees of consistency that can be roughly characterized as follows. Degree 0 protects transactions from overwriting each others uncommitted updates. Degree 1 adds

totality. Degree 2 adds protection against reading the uncommitted updates of others. Degree 3 provides serial consistency. Further details are provided in the taxonomy [Gray et al. 76].

The motivation for using lower degrees of consistency is that they permit more concurrency, perform better because fewer locks are set, and are less complex to implement than higher degrees. A transactional storage system could support lower degrees of consistency. However, there are a number of ways of achieving the benefits of lower degrees of consistency with the system as described. For example, if a transaction runs for a long time it could be broken up into many smaller transactions.

3.3.4 Broken Read Locks

In some cases when a data item that a transaction has read becomes obsolete it is not necessary to abort the transaction. A common example is a cache manager that holds data as a convenience. If one of the data items it is holding is no longer current it would be content to be informed of that fact so it could remove the item from its cache, and then proceed.

Transactional storage could inform clients that a data item they previously received has become obsolete, and give them the option of not aborting their transaction. This is called breaking a read lock, in reference to the read lock the data item held. The advantage of the scheme is that fewer transactions are aborted.

3.3.5 Releasing Read Locks

Transactional storage could allow clients to inform it that although they read a data item, it did not influence their computation, and its corresponding read lock could be released. As a general rule, the more read locks a transaction holds, the more likely it is to conflict with another transaction and be aborted. In addition, the system must verify that read locks are still being held at commit time, which increases the amount of communication that must occur.

3.3.6 Deadlock

Cyclic dependencies, or deadlocks, can occur when clients dynamically acquire resources. For example, if client A acquires X and wants Y, and client B acquires Y and wants X, a deadlock exists. In order to guarantee progress either A or B must be aborted and the other allowed to proceed.

Deadlock prevention requires that all transactions either declare in advance what resources they are going to use, or acquire resources in a fixed order. Deadlock prevention is usually considered too restrictive for general use, and it reduces concurrency by locking too much for too long.

Deadlock detection maintains a resource dependency graph to detect when a cyclic dependency occurs. This approach is difficult to efficiently implement in a decentralized system, although some proposals have been made [Gray 78; Obermarck 80].

A time-out strategy resolves deadlocks by aborting a transaction if it holds a resource for too long that another transaction has requested. The problem with a time-out scheme is that in a system with a lot of contention for resources, once a transaction has run longer than the timeout interval it is likely to be aborted. This problem can be alleviated somewhat by specifying the time-out interval for a transaction when it is created.

3.3.7 Checkpoints and Save Points

It is possible to commit a transaction and still keep the transaction active for further operations. Such a commit is called a *checkpoint.* A checkpoint allows a transaction to save a consistent set of results to hedge against the possibility that it might not complete [Gray et al. 79].

A *save point* is an intermediate place in a transaction that recovery management can return to if the transaction gets into trouble. For example, if a transaction is involved in a deadlock, it may be possible for recovery management to return to the transaction's last save point. If this is possible, all of the transaction's work will not be discarded, as would be the case if recovery management could only abort the entire transaction.

3.3.8 Nested Transactions

The one level transaction structure that has been presented can be extended to include nested transactions [Davies 73; Reed 78]. Nested transactions are just like normal transactions, except that when they commit their results are only visible to their parent transaction. For example, imagine T has two nested transactions, T1 and T2, running at the same time. T1 and T2 can not detect each others presence, and their results are only visible to T when they commit. The results of T1 and T2 are made public when T commits.

Nested transactions may provide a performance improvement by making transactions less likely to abort. For example, in Section 3.2.4 we described a scheme that forced a transaction to abort if a processor failed. If Enter-Door created a separate nested transaction for each remote evaluation, then on processor failure it could abort the corresponding nested transaction and retry the evaluation.

3.3.9 Stable Storage Failure

The fundamental premise that we are operating under is that stable storage will not fail. However, stable storage will fail sometime. If stable storage fails it is the clients' responsibility to reconstruct its contents. Simply using an obsolete copy of the destroyed data will not do. Assume that there is a catastrophic failure of volume V, and all that remains is an out of date copy of V made at time T. One can not simply restore V to its state at time T because consistency would be destroyed. Consider the case where V stores bank account balances, and substantial withdrawals have been made since time T. Restoring V would result in the bank giving away money.

Obsolete information can be of use, however. For example, if V and V's log from time T to the present are available stable storage can be totally reconstructed. If a log is not available, a salvaging program could read obsolete data and combine it with information available from outside the system to rebuild storage.

3.4 Summary

Transactional storage was introduced as an ideal way of allowing stable storage to be shared between clients. The fundamental characteristics of transactional storage were described, along with a model set of objects and primitives. A survey of existing and proposed algorithms for the implementation of transactional storage was provided. The final section of the chapter outlined some refinements that can be made to the model and its implementation.

Exercise

1. Give an algorithm for TryToSet. Will your algorithm always terminate in finite time?

Chapter 4: Location

In many instances objects naturally physically move about in a decentralized system. A removable storage volume is an obvious example. Other objects, such as coordinators, can also move if the processor that implements them changes.

This chapter describes the techniques that we employ to determine where an object is implemented. The first section introduces the notion of an index, the second section shows how indexes can be used to implement indirection, the third section introduces indirect references, the fourth section discusses how files, volumes, and other objects are located, and the last section describes choice references. The chapter ends with a brief summary.

4.1 Indexes

An *index* holds a set of *entries*. An entry is an array of bytes, and each entry is named with an *entry-name*. Like other objects, indexes are named by references, and Open is evaluated to obtain a class to service an index. Like files and volumes, indexes can be made immutable. The following functions define the interface to indexes:

Create-Index[storage: File / index: Index]

> Create-Index takes a file reference and returns an uncounted reference for a new index. It is presumed that *storage* either contains the data structures that represent an index, or is a zero length file. We will see in Chapter 6 why *storage* might already contain the data structures of an index. The index will inherit the unique identifier of *storage*. An index reference is defined to have the following type:
>
> Index ← Extend[Record[storage: File], Reference];

ic: Index-Class | Write[entry-name: Byte-Array, value: Byte-Array]

> Write enters an entry in an index and names it entry-name. If an entry previously existed under the specified entry-name its value is updated. If an entry previously existed and the value argument to Write is NIL, the entry is deleted from the index.

ic: Index-Class | Read[entry-name: Byte-Array / value: Byte-Array]

> Read returns the entry named by entry-name. If there is no entry under the specified name, NIL is returned.

Entry ← Record[entry-name: Byte-Array, value: Byte-Array];

ic: Index-Class | Enumerate[last: Entry / next: Entry]

> Enumerate allows all of the entries in an index to be enumerated. The first entry in an index is returned when last is NIL, and next is NIL when there are no more entries.

There are many algorithms for implementing indexed files. Popular approaches are presented by [Knuth 73] and [McCreight 77].

4.2 Indirection

4.2.1 Model

As the first step to implement location, we introduce processor-independent indirection. Indirection is processor-independent in the sense that an indirect pointer can be made on one processor and dereferenced on another processor. The following four functions define the interface to the indirection mechanism:

Create-Indirect[record: Record, index: Index, tc: TC / ie: Indirect]

> Create-Indirect creates an indirect entry, places *record* in the entry, and returns an indirect pointer to the new entry.

Lookup[ie: Indirect, tc: TC / record: Record]

> Lookup returns the contents of the indirect entry specified by *ie.*

Change-Indirect[ie: Indirect, record: Record, tc: TC]

> Change-Indirect places *record* in the indirect entry specified by *ie.* If *record* is NIL, the indirect entry is deleted.

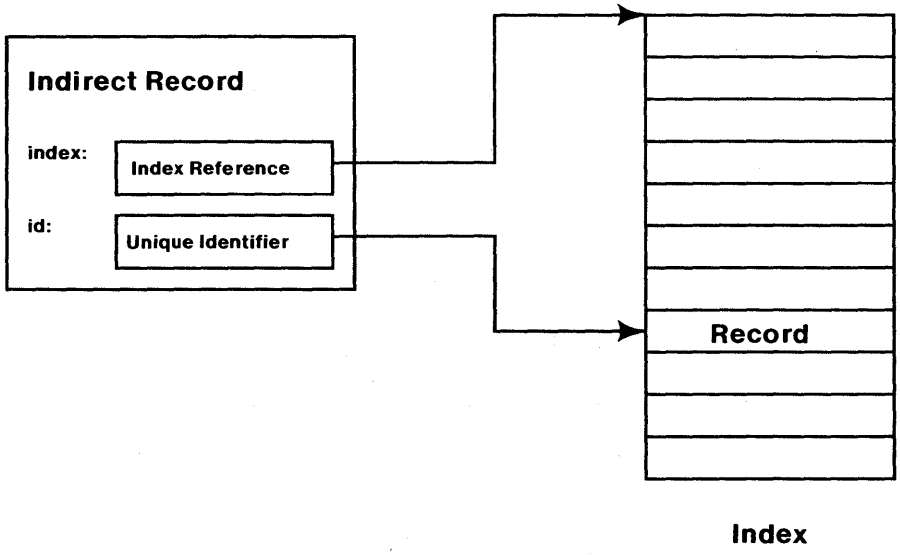
Normalize[ie: Any, tc: TC / record: Record]

> If *ie* is indirect, Normalize dereferences *ie* using Lookup until the result is not an indirect record.

4.2.2 Basic Algorithm

Indexes are used to implement indirection. As shown in Figure 4.1, an indirect record points to the index entry that is used to hold its value.

```
Indirect ← Record[indirect-id: UniqueID, index: Index];

Create-Indirect[record: Record,  index: Index, tc: TC / ie: Indirect]
  ← Prog[ [ic: Index-Class];
      ie ← Create[Indirect];
      ie.indirect-id ← GetUniqueID[];
      ie.index ← index;
      ic ← Open[index, tc];
      IF Is[ic, Error-Type] THEN Return[ic];
      -- check for unique id generator malfunction
      IF Not[Null[ic | Read[ie.indirect-id]]] THEN [
            ic | Close[];
            Return[Error['UniqueIDMalfunction]];
            ];
      ic | Write[ie.indirect-id, Encode[record]];
      ic | Close[];
      Return[Add-Type[ie, Major-Type[record]]];
      ];

Lookup[ie: Indirect, tc: TC / record: Record] ← Prog[
      [ic: Index-Class];
      ic ← Open[ie.index, tc];
```



An Indirect Record

Figure 4.1

```
        record ← Decode[ic | Read[ie.indirect-id]];
        ic | Close[];
        Return[record];
        ];
    Change-Indirect[ie: Indirect, record: Record, tc: TC] ← Prog[
        [ic: Index-Class; contents: Any];
        contents ← IF Null[record] THEN NIL ELSE Encode[record];
        ic ← Open[ie.index, tc];
        ic | Write[ie.indirect-id, contents];
        ic | Close[];
        ];
    Normalize[ie: Any, tc: TC / record: Record] ← Prog[ [];
        record ← ie;
        UNTIL Not[Is[record, Indirect]] DO record ← Lookup[record, tc];
        Return[record];
        ];
```

4.3 Indirect References

With the structure we have described so far once a reference is given to a client it must never change. There are many reasons to eliminate this restriction. For example, if a located reference could change, then the location of its referent could change. At times it may also be desirable to change the referent of a reference, perhaps to cause clients to use a new object.

To allow references to change, we introduce the notion of an indirect reference. Indirect references are created by applying Create-Indirect to a reference. Indirect references are used just like ordinary references. They are counted or uncounted, depending on the type of their original reference. CopyReference and DestroyReference are used to copy and destroy indirect references. Indirect-References take up less space than normal references, and thus can reduce the amount of storage occupied by common references.

ChangeReference is used to change an indirect reference and destroy the old reference that was in its index entry. Change-Indirect can be used to change an indirect reference, but it does not destroy the old reference.

ic: Indirect-Class | ChangeReference[nref: Reference]

> ChangeReference causes the indirect reference represented by *ic* to be changed to point at *nref*. *ic* is also changed to service *nref*.

When a counted-indirect reference is made for an object, the object's reference count is increased by one. When the destruction of an indirect reference results in the destruction of its referent (it was the only counted reference), then the indirect reference's index entry is also destroyed. Thus, if three counted-indirect references share an indirect entry, the indirect entry will be destroyed when the last of the three references is destroyed.

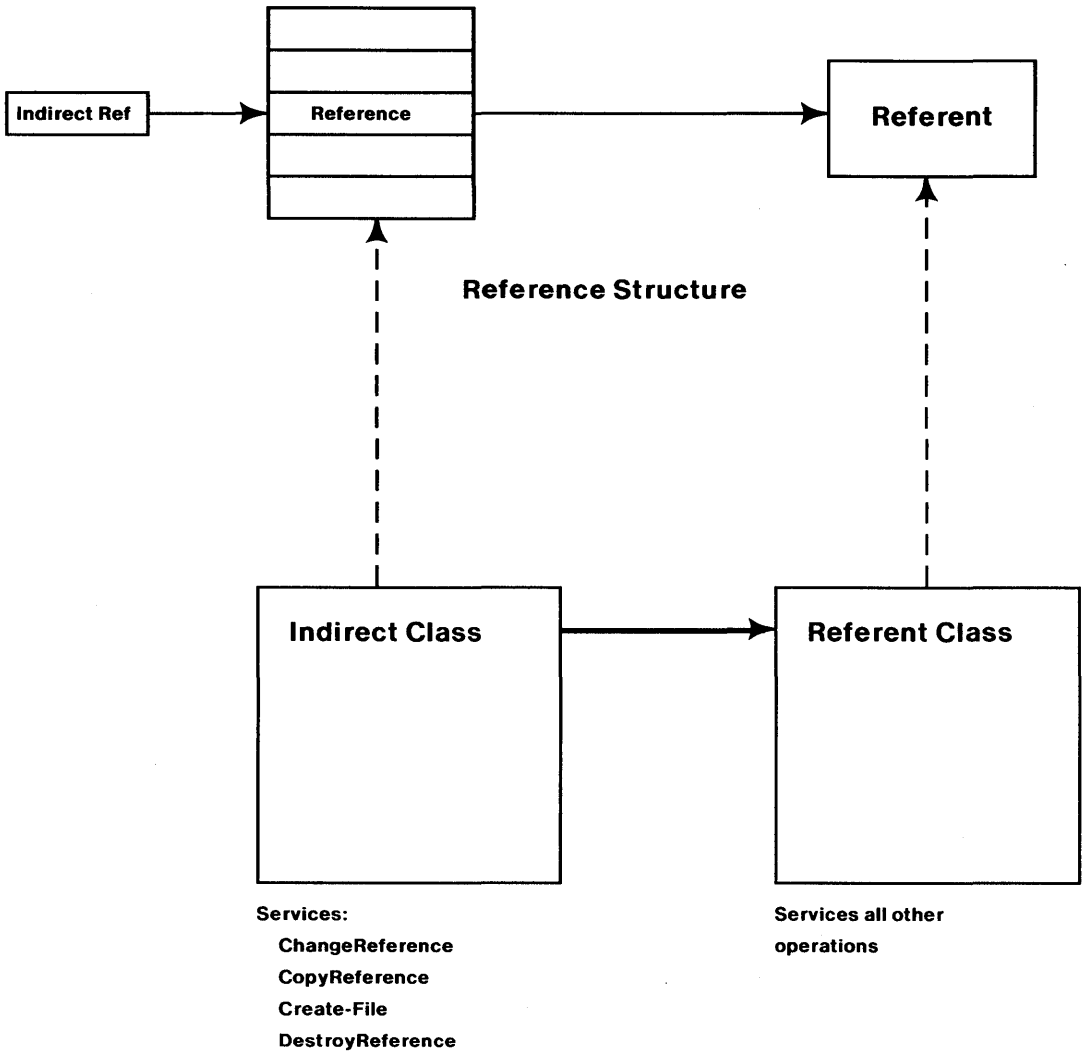
A file reference is defined to include a reference for its containing volume. Thus, when a file is created on an indirect volume, a reference for the indirect volume is included in the new file reference.

Figure 4.2 shows the class structure that is established when an indirect reference is opened.

All requests are first considered by a class specialized for indirect references, which is called an indirect class. If a request is not one of the four operations that the indirect class handles, then it is passed along to a class for the indirect reference's referent. A similar class structure will be used for other types of references.

```
Open-Indirect[ref: Reference, tc: TC, ring: List[Key], guards: List[Key] / c: Class]
        ← Prog[ [d-ref: Reference];
        d-ref ← Lookup[ref, tc];
        c ← Open[d-ref, tc, ring, guards];
        IF Is[c, Error-Type] THEN Return[c];
        c ← Create-Class[List[
                'CopyReference, 'Default-Copy,
                'ChangeReference, 'Indirect-Change,
                'DestroyReference, 'Indirect-Destroy,
                'Create-File, 'Indirect-Create-File],
                c];
        -- Instance variables: ref, tc, ring, guards
        Return[c];
        ];

Indirect-Change[nref: Reference] ← Prog[ [];
        -- change reference
        -- first, destroy old reference
        superclass | DestroyReference[];
        superclass | Close[];
        -- second, update index
        Change-Indirect[ref, nref, tc];
        -- third, open new reference and switch superclass
        superclass ← Open[nref, tc, ring, guards];
        IF Is[superclass, Error-Type] THEN Return[superclass];
        ];

Indirect-Destroy[/ deleted: Boolean] ← Prog[ [];
        -- destroy reference pointed to
        deleted ← Apply[superclass, request];
        -- destroy indirect if object deleted
        IF deleted THEN Change-Indirect[ref, NIL, tc];
        Return[deleted];
        ];

Indirect-Create-File[] ← Prog[ [];
        -- substitute an indirect volume ref
        file-ref ← Apply[superclass, request];
        IF Is[file-ref, Error-Type] THEN Return[file-ref];
        file-ref.volume ← self | CopyReference[];
        ];
```

Open-Indirect can cache the results of Lookup requests. However, if it uses one of these cached references and an Error results, then it must use Lookup to determine if its cached value is still current. If an error is going to occur it will occur at Open time, so it is always safe for Open-Indirect to attempt to use an obsolete cache entry.



Class Structure

Structure of an Indirect Class

Figure 4.2

4.4 Object Location

Indirect references are used to implement location for volumes, coordinators, and indexes. As a matter of convention, indirect references are always made for objects that might move. When the location of such an object changes, Change-Indirect is used to update the object's indirect entry with a new located reference.

Take the case of a removable volume as an example. When the volume is created, an indirect reference for it is made. This indirect reference is what is given to clients and passed around in the system. Whenever the removable volume is mounted, its location is placed in the index specified by its indirect reference. Clients that hold the volume's reference will then be able to access it.

Indexes can not always be located with indexes because of the obvious problem with recursion. The index-index is introduced to solve this problem. The index-index is named by a fixed, located reference. Because the index-index will be used frequently an implementation should represent its reference compactly. The reference for the index-index is a located reference:

Index-Index: Located-Index

The location in the index-index reference is *Broadcast.* This is the only use of the broadcast address we will make. We assume that every processor that listens for broadcasts knows of a processor that will service requests for the index-index.

Files and indexes are assumed to be located at the same place as their containing volume, and we can take advantage of this fact to reduce our need for indirection. We simply determine the location of an index's or file's volume, and use it to construct a located reference.

```
Open-File[ref: Reference, tc: TC, ring: List[Key], guards: List[Key] / c: Class] ← Prog [
        [volume-ref: Reference];
        -- Get located volume reference
        volume-ref ← Normalize[ref.volume, tc];
        IF Not[Is[volume-ref, Located]] THEN Return[Error['NotFound]];
        -- Open file at the processor where the volume is
        c ← Open[Create-Located[ref, volume-ref.loc], tc, ring, guards];
        IF Is[c, Error-Type] THEN Return[c];
        Return[Create-Class[List['CopyReference, 'Default-Copy], c]];
        ];

Open-Index[ref: Reference, tc: TC, ring: List[Key], guards: List[Key] / c: Class] ← Prog [
        [volume-ref: Reference];
        -- Get located volume reference
        volume-ref ← Normalize[Normalize[ref.file, tc].volume, tc];
        IF Not[Is[volume-ref, Located]] THEN Return[Error['NotFound]];
        -- Open index at the processor where the volume is
        c ← Open[Create-Located[ref, volume-ref.loc], tc, ring, guards];
        IF Is[c, Error-Type] THEN Return[c];
        Return[Create-Class[List['CopyReference, 'Default-Copy], c]];
        ];
```

4.5 Choice References

Let us return to the way we name objects for a moment. We have stated that a reference is an unambiguous name for a single object. Here we modify that view slightly. Imagine that we wanted

to use some object from the list O1 ... On, but we did not care which object was chosen. To implement this idea we introduce the idea of a choice reference. A choice reference is a set of references S, and when a choice reference is opened, the class returned will correspond to one of the elements of S.

A strategy could be employed to select which element of a choice reference is selected. For example, in the case of a volume choice reference we could select the volume with the largest remaining capacity.

An application for choice references is when any one of a number of processors can service requests for an object. In this case, we do not want to place the address of a single processor in the object's index entry. Instead, we create a processor choice reference that includes all of the processors that can service the object, and use this choice reference as the object's location. A choice reference is created by Create-Choice:

```
Create-Choice[choices: List[Reference] / cr: Choice]
```

Create-Choice creates a reference whose referent can be any one of the references in choices.

```
Choice ← Record[choices: List[Reference]];

Create-Choice[choices: List[Reference] / cr: Choice] ← Prog[ [];
        cr ← Create[Choice];
        cr.choices ← choices;
        Return[cr];
        ];
```

The model implementation for a choice reference opens all of its possible choices at once, and uses the first object to respond. Objects that are not selected are closed.

```
Open-Choice[ref: Reference, tc: TC, ring: List[Key], guards: List[Key] / class: Class]
    ← Prog[ [choice-made: Condition-Variable, choice-lock: Lock];
        class ← NIL;
        choice-lock ← Create-Lock[];
        choice-made ← Create-Condition-Variable[];
        MapCar[ref.choices, 'Choice-Open];
        UNTIL Not[Null[Get-Choice-Class]] DO Wait[choice-made];
        Return[Get-Choice-Class[]];
        ];

Get-Choice-Class[/c: Class] ← Critical[choice-lock, 'class];

Set-Choice-Class[c: Class] ← Critical[choice-lock, 'Prog[ [];
        IF Is[c Error-Type] THEN Return[];
        IF Null[class] THEN [
                class ← c;
                Broadcast[choice-made];
                ]
        ELSE c | Close[];
        ]];

Choice-Open[ref: Reference] ← Fork['Set-Choice-Class[Open[ref, tc, ring, guards]]];
```

4.6 Summary

This chapter described a general framework for locating objects. Indexes were defined, and used to implement indirect records. Indirect records in turn allowed indirect references to be created. Indirect references were used to keep the location of an object in a single place that can be conveniently updated. Choice references were introduced to handle a service that is implemented by many objects.

Exercise

1. Under what circumstances would you use a choice reference for a volume?

Chapter 5: Replication

So far we have described a system where only one copy is kept of files, indexes, and volumes. This chapter describes how objects can be replicated to improve their availability, reliability, and performance. The first section describes a model of the replication facilities we provide, the second section shows how our model can be implemented, the third section argues that the replication algorithm preserves the properties of transactions, the fourth section offers some refinements to the basic algorithm, and the final section compares our method of replication with previous work. The chapter ends with a brief summary.

5.1 Suites

To introduce what a suite is and how one works let's start with an example. Imagine that we would like to maintain three copies of a file. We will say a copy is *current* if it has received all of the updates that have been applied to the file. Here are three approaches for maintaining the copies:

1. Always update all three copies. Thus, all three copies will always be current. To read the file, we could read any one of its three copies. Of course, if any one of the three copies is unavailable we would be unable to update the file.

2. Always update at least two copies. Both of these copies must be current, to guarantee that a current copy always receives all of the updates that are applied to the file. To read, we know that if we examine two copies, one of them is guaranteed to be current. If version numbers are kept on the copies, then we can identify a current copy, because it will have the highest version number.

3. Always update at least one copy. Once again, this copy must be current, to guarantee that a current copy always receives all of the updates that are applied to the file. To read, we know that if we examine all three copies, one of them is guaranteed to be current. Version numbers are once again required to determine which of the three copies is current.

These methods all preserve the invariant that every set of copies that they write intersects with every set of copies that they read. In this way they are always guaranteed to provide current information.

We will call a collection of copies that implements a single object a *suite*, and individual copies *representatives*. Our example described a file suite with three representatives. Furthermore, we will call a set of representatives that is examined to read a suite a *read quorum*, and a set of representatives that is written in response to a suite write operation a *write quorum*. As in our example, every read quorum and every write quorum must have a representative in common. The read and write quorums described by the above three methods are respectively:

1. Write Quorum: {1, 2, 3}
   Read Quorums: {1}, {2}, {3}

2. Write Quorums: {1, 2}, {1, 3}, {2, 3}
   Read Quorums: {1, 2}, {1, 3}, {2, 3}

3. Write Quorums: {1}, {2}, {3}
   Read Quorum: {1, 2, 3}

The basic idea behind our replication mechanism is a compact encoding of a suite's read and write quorums. Every representative of a suite is assigned a non-negative integer, which represents a number of votes. A read quorum is considered to be any set of representatives whose votes total to *r*, and a write quorum is considered to be any set of representatives whose votes total to *w*. Non-negative integers *r* and *w* are chosen such that $r + w$ is greater than the total number of votes assigned to the suite. A suite's voting configuration is the triple <vote-assignment, r, w>.

As in our example, when updates are applied to a write quorum every member of the write quorum must be current. Thus, every read quorum and every write quorum have a representative in common, and every read quorum will always have a representative that is current. Version numbers make it possible to identify this representative. Section 5.3 describes the algorithm in detail.

To read a suite, at least *r* votes worth of representatives must be available. To write a suite, MAX[r,w] votes worth of representatives must be available. This is because before a suite can be written a read quorum of representatives must be assembled to determine what version number a current representative will have. When the contents of a suite are totally replaced only *w* votes worth of representatives need be available (Section 5.4.6).

The algorithm has a number of desirable properties. It starts with and preserves the properties of transactional storage, including totality, serial consistency, and external consistency (Section 3.1.2). It continues to operate even when some representatives are unavailable. It is simple, and can be used to create many types of suites. In addition, all of the copies of an object, including temporary copies that clients create to increase performance, can be incorporated into the framework.

A suite's characteristics can be chosen from a range of possibilities by adjusting its voting configuration. For example, heavily weighting high performance representatives will result in a suite with higher performance, and heavily weighting representatives that are very reliable will result in a more reliable suite. A completely decentralized structure results from equally weighting representatives, and a completely centralized scheme results by assigning all of the votes to a single representative.

Once the general reliability and performance of a suite is established by its voting configuration, the relative reliability and performance of Read and Write can be controlled by adjusting *r* and w. As *r* decreases, reads become more efficient and reliable. As *w* decreases, writes become more efficient and reliable. The choice of *r* and *w* will depend on the read to write ratio expected, the relative costs of reading and writing, and desired suite characteristics.

Table 5.1 suggests the diverse mix of properties that can be created by appropriately setting *r* and *w*. The blocking probabilities shown in the table represent the probability that a quorum will not be available. We have assumed that the probability that a representative is unavailable is .01.

Example 1 is configured for a file with a high read to write ratio in a single server, multiple user environment. Replication is used to enhance the performance of the system, not the reliability.

There is one server on a local network that can be accessed in 75 milliseconds. Two users have chosen to make copies on their personal disks by creating weak representatives, or representatives with no votes (see Section 5.5.1 for a complete discussion of weak representatives). This allows them to access the copy on their local disk, resulting in lower latency and less traffic to the shared server.

Example 2 is configured for a file with a moderate read-to-write ratio that is primarily accessed from one local network. The server on the local network is assigned two votes, with the two servers on remote networks assigned one vote apiece. Reads can be satisfied from the local server, and writes must access the local server and one remote server. The system will continue to operate in read-only mode if the local server fails. Users could create additional weak representatives for lower read latency.

Example 3 is configured for a file with a very high read to write ratio, such as a system directory, in a three server environment. Users can read from any server, and the probability that the file will be unavailable is very small. Updates must be applied to all copies. Once again, users could create additional weak representatives on their local machines for lower read latency.

	Example 1	Example 2	Example 3
Latency (msec)			
Representative 1	75	75	75
Representative 2	65	100	750
Representative 3	65	750	750
Voting Configuration	<1, 0, 0>	<2, 1, 1>	<1, 1, 1>
r	1	2	1
w	1	3	3
Read			
Latency (msec)	65	75	75
Blocking Probability	1.0×10^{-2}	2.0×10^{-4}	1.0×10^{-6}
Write			
Latency (msec)	75	100	750
Blocking Probability	1.0×10^{-2}	1.0×10^{-2}	3.0×10^{-2}

Table 5.1

Create-Suite is used to organize a collection of objects into a suite.

Create-Suite[id: UniqueID, r: Integer, w: Integer, rep: List[Reference], votes: List[Integer] / suite: Suite]

Create-Suite creates a suite reference from a specification that consists of the identifier to be assigned to the suite, r, w, a list of representative references, and a list of the representatives' respective vote assignments. The suite reference returned can be opened and used like an ordinary object. The suite's type is determined by the type of its representatives. All of the references in *rep* must be of the same type, and they all must have the same version number. If the resulting reference is a volume reference, then files created on *suite* will be file suites.

Suite ← Record[];

```
Normal-Suite ← Extend[Suite, Record[id: UniqueID, r: Integer, w: Integer, rep:
                List[Reference], votes: List[Integer]];

Create-Suite[id, r, w, rep, votes / suite] ← Prog[ [];
        -- start making up the suite
        suite ← Create[Normal-Suite];
        suite.r ← r; suite.w ← w; suite.votes ← votes;
        suite.id ← id; suite.rep ← rep;
        Return[Add-Type[suite, Major-Type[car[rep]]]];
        ];
```

When Create-Suite is applied to a set of volumes a volume suite results. A volume suite is a template for creating file suites. When Create-File is applied to a volume suite a file suite will be created that has the same voting configuration as the volume suite. The new file suite will have a set of representatives whose votes total to at least MAX[r,w]. This ensures that the new suite will be fully functional.

In order to allow the replication algorithm to keep track of which representatives have current information, and to allow the algorithm to update obsolete representatives, all objects that will be used as representatives must implement the following operations.

class: Class | GetVersion[/ version: Integer]

> The version number of an object is a count of the number of writes that have been performed on the object. Consistency is guaranteed for version numbers. In other words, once GetVersion returns the version number of an object, no other transaction will update the object before the the transaction associated with *class* ends.

class: Class | Copy[copy-from: Reference]

> Copy copies the index or file specified by *copy-from* to the index or file serviced by *class*. Copy is implemented in such a way that an object can be copied to itself. After Copy finishes, the original object and its copy are indistinguishable.
>
> The object serviced by *class* receives the following state from *copy-from*: version number, reference count, length, immutability, and contents. The last thing that is copied is the version number. Thus, if a transaction is committed before Copy finishes, the object serviced by class will still have its old version number.
>
> In order to guarantee that an object has a monotonically increasing version number, *copy-from* must have a version number that is larger or equal to the version number of *class*. If this is not the case, Copy will return Error['VersionError].

5.2 Basic Algorithm

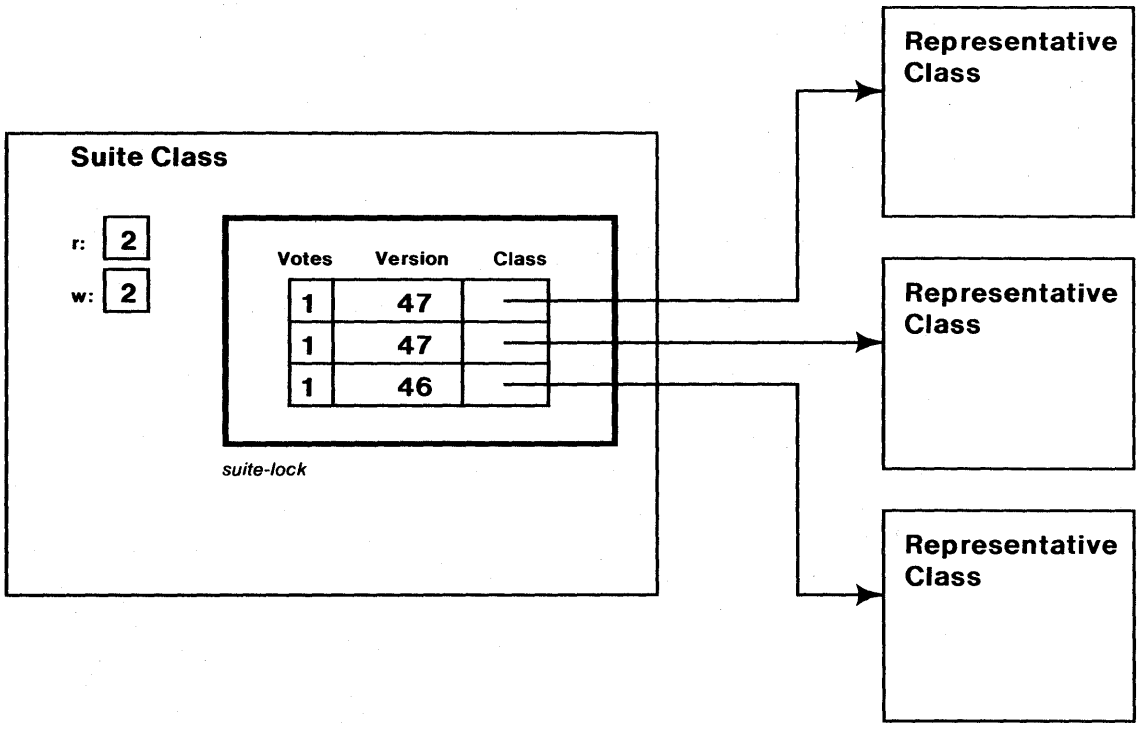
We present the basic algorithm by describing how operations on a suite are transformed into operations on the suite's representatives. The perspective taken is that of a single transaction. Of course there can be many transactions accessing a given suite at the same time, all performing the algorithm.

We also provide a step by step synthesis of a class to implement the algorithm. The class is initialized by Open-Suite, which is called by Open to create a class to service a suite. Before Open-Suite returns, it gathers a read quorum to determine the suite's version number. Open-Suite

establishes the correspondence between operations on the suite (e.g. Create-File) and the functions that implement these operations in terms of the suite's representatives (e.g. Suite-Create).

Careful use of critical sections has been made so the class can process simultaneous requests. Figure 5.1 shows the gross structure of a suite class.

```
Open-Suite[input-ref: Suite, tc: TC, ring: List[Key], guards: List[Key] / sc: Class] ← Prog[
    [x: List[Representative];
    r: Integer;  w: Integer;  votes: List[Integer];
    number-of-representatives: Integer;
    -- normal form of input-ref is suite-ref
    suite-ref: Normal-Suite;

    -- Broadcast[crowd-larger] occurs when a new representative becomes available
    crowd-larger: ConditionVariable;

    -- write-lock must be locked to access write-quorum or raw-votes
    write-lock: Lock;
    write-quorum: List[Representative];
    raw-votes: Integer;
    -- suite-lock must be locked to access any of the following:
    suite-lock: Lock;
    suite: List[Representative];
    current: Integer;          -- a current representative has this version number
    not-found-votes: Integer;
    ];
    -- let's see what is out there and initialize suite variables
    Initiate-Inquiries[];
    x ← Collect-Read-Quorum[];
    -- if suite was not found, return error to client
    IF Is[x, Error-Type] THEN Return[x];
    -- Instance variables: r, w, votes, number-of-representatives, suite-ref, crowd-larger,
    write-lock, write-quorum, raw-votes, suite-lock, suite, current, not-found-votes,
    input-ref, tc, ring, guards
    Return[Create-Class[List[
            'Read, 'Suite-Read,
            'Copy, 'Suite-Copy,
            'Enumerate, 'Suite-Read,
            'GetVersion, 'Suite-Read,
            'Write, 'Suite-Write,
            'IsImmutable, 'Suite-Read,
            'SetImmutable, Suite-Write,
            'GetSize, 'Suite-Read,
            'GetID, 'Suite-GetID
            'SetSize, 'Suite-Write,
            'GetTransactionClass, 'Suite-Read,
            'CopyReference, 'Suite-CopyRef,
            'DestroyReference, 'Suite-Write,
            'Close, 'Suite-Close,
            'Create-File, 'Suite-Create], NIL]];
    ];
```



Structure of a Suite Class

Figure 5.1

```
Suite-GetID[/ id: UniqueID] ← [suite-ref.id];
```

A quorum is represented by a list of Representative records. Collect-Quorum and Collect-Write-Quorum collect quorums. They are described in detail later.

```
Representative ← Record[
        version: Integer,
        votes: Integer,
        class: Class,
        ref: Reference];
```

A suite read operation is an operation that examines the state of a suite without changing it. The operations Read, Enumerate, IsImmutable, GetSize, GetTransactionClass, and GetVersion are all transformed into Suite-Read.

A suite read operation can be performed by any current representative. To find a current representative the algorithm first collects a read quorum. From this quorum a current representative is selected, and the requested read operation is performed by this representative. Ideally, one would like to read from the representative that will respond the fastest.

If storage is damaged by an unexpected error, Error['StorageDamaged] will be returned by a read request. In this event Suite-Read could attempt to find another current representative to read from.

```
Suite-Read[/ result: Any] ← Prog[ [];
        result ← Apply[Select-Current[Collect-Quorum[r]].class, request];
        Return[result];
        ];
Select-Current[quorum: List[Representative] / rep: Representative] ← Critical[suite-lock,
'Prog[ [];
        FOR rep IN quorum DO [
                -- find a current representative
                IF rep.version=current THEN
                        Return[rep];
                ];
        ]];
```

A suite write operation is an operation that updates the state of a suite and creates a new version. The operations Write, SetImmutable, SetSize, and DestroyReference are transformed into Suite-Write.

A suite write operation is performed by a write quorum, all of whose members are current. The algorithm first gathers a write quorum, and then every representative in the quorum performs the requested write operation. After all of the members of the quorum have finished, a check is made to ensure that none of them returned an error, and the result from one of the representatives is returned as the result of the write operation.

The result of concurrent writes that update the same portion of a suite is undefined. In the model implementation, if two concurrent writes update the same portion of a suite it is possible that half of the suite will assume one value, and the other half of the suite a different value. When the suite is subsequently read, it will be indeterminate which of these two values will be returned.

```
Suite-Write[/ result: Any] ← Prog[ [];
        Return[Apply-Quorum[Collect-Write-Quorum[]]];
        ];

Apply-Quorum[quorum: List[Representative] / result: Any] ← Prog[
        [process: List[Process]; x: Any];
        -- start all representatives working on the request
        process ← MapCar[quorum, 'Fork-Request];
        -- wait for all of the representatives to finish
        result ← MapCar[process, 'Join];
        FOR x IN result DO
                IF Is[x, Error-Type] THEN Return[x];
        Return[car[result]];
        ];

Fork-Request[rep: Representative / p: Process] ←
        Fork['Apply[rep.class, request]];
```

A Copy request is a special write operation. Because the contents of the suite are being replaced, all of the members of the write quorum do not have to be current. A Copy operation is transformed into Suite-Copy. Suite-Copy first replaces the contents of the suite, and then updates cached version numbers.

```
Suite-Copy[/ result: Any] ← Critical[write-lock, 'Prog[
        [rep: Representative];
        -- replace a write quorum
        write-quorum ← Collect-Quorum[w];
        result ← Apply-Quorum[write-quorum];
        IF Is[result, Error-Type] THEN Return[result];
        Mark-Reps-Current[write-quorum];
        Return[result];
        ]];

Mark-Reps-Current[quourm: List[Representative]] ← Critical[suite-lock, 'Prog[
        [rep: Representative];
        -- update cached version numbers
        current ← car[write-quorum].class | GetVersion[];
        FOR rep IN write-quorum DO
                rep.version ← current;
        ]];
```

A CopyReference request is a write operation, because it can update the state of the suite by incrementing its reference count. A CopyReference operation is transformed into Suite-CopyRef.

```
Suite-CopyRef[counted: Boolean / nr: Suite] ← Prog[
        [];
        -- copy the references of a write quorum
        Apply-Quorum[Collect-Write-Quorum[]];
        -- return a copy of the reference
        nr ← Copy[input-ref];
        IF counted THEN Add-Type[nr, Counted]
                ELSE Remove-Type[nr, Counted];
```

```
        Return[nr];
        ];
```

As we discussed in the last section, when Create-File is applied to a volume suite a file suite is created. Representatives for the new file suite are created on a set of volume representatives whose votes total to at least MAX[r,w]. The model implementation of Suite-Create does not allow a volume suite to include representatives that are reconfigurable or protected, concepts covered in Chapters 6 and 7 respectively.

We digress for a moment to explain a fine point about the operation of Suite-Create. There are two types of file suite references. A suite reference returned from Create-Suite is fully expanded, and contains a list of a suite's representatives, vote assignments, and so on. This type of suite reference is said to be in normal form. The file suite reference returned from Suite-Create does not contain a reference for each of its representatives. Such references are condensed and consist of the unique identifier of the suite's representatives and a volume suite. This type of file suite reference can be easily reconstituted to normal form, as we shall see later.

```
Suite-Create[id: UniqueID / nr: Suite-Reference] ← Prog[
        [result: Any];
        result ← Apply-Quorum[Collect-Quorum[Max[r, w]]];
        IF Is[result, Error-Type] THEN Return[result];
        -- create file reference
        nr ← Create[Extend[File, Suite]];
        nr.id ← id;
        nr.volume ← self | CopyReference[];
        -- return the new suite's reference.
        Return[nr];
        ];
```

This concludes our discussion of how suite operations are transformed into operations on suite representatives. We now turn our attention to how a suite is initialized, and how quorums are gathered.

When a suite is opened queries are sent out to determine the version numbers of the suite's representatives. If a representative is available, it responds with its version number, and it can be considered for inclusion in a quorum. As each representative reports its version number the highest version number seen is updated. After a read quorum of representatives have reported their version numbers the highest version number that has been seen is the highest that exists. This in essence is the version number of the suite, and any current representative will have this version number. If a read quorum of a suite's representatives do not exist, provisions are made so clients are told that the suite does not exist.

It may be that a version number inquiry will never return because its corresponding representative is unavailable. As stated in Section 3.1.2, outstanding uncompleted version number reads do not affect the ability of a transaction to commit.

```
Initiate-Inquiries[] ← Prog [ [i: Integer];
        -- initialize suite
        suite-ref ← Expand-Suite[input-ref];
        r ← suite-ref.r;
        w ← suite-ref.w;
```

```
        votes ← suite-ref.votes;
        number-of-representatives ← Length[suite-ref.rep];
        write-lock ← Create-Lock[];
        write-quorum ← NIL;
        suite-lock ← Create-Lock[];
        crowd-larger ← Create-ConditionVariable[];
        suite ← NIL;
        current ← 0;
        not-found-votes ← 0;
        -- send out version number inquiries
        FOR i FROM 1 TO number-of-representatives DO
            Fork['Open-Representative[i]];
        ];
    Open-Representative[i: Integer] ← Prog[
        [rep: Representative];
        rep ← Create[Representative];
        rep.ref ← Nth[suite-ref.rep, i];
        rep.votes ← Nth[suite-ref.votes, i];
        rep.class ← Open[rep.ref, tc, ring, guards];
        IF rep.class=Error['NotFound] THEN Not-Found[rep.votes];
        IF Is[rep.class, Error-Type] THEN Return[];
        rep.version ← IF Is[rep.ref, Volume] THEN 0 ELSE class | GetVersion[];
        Note-Representative[rep];
        ];
    Note-Representative[rep: Representative] ← Critical[suite-lock, 'Prog[ []];
        -- a representative has responded to our inquiry
        IF rep.version > current THEN current ← rep.version;
        suite ← Append[suite, rep];
        Broadcast[crowd-larger];
        ]];
    Not-Found[votes: Integer] ← Critical[suite-lock, 'Prog[ []];
        -- a representative was not found
        not-found-votes ← not-found-votes + votes;
        IF not-found-votes<w THEN Return[];
        -- can never get a read quorum; wake up Collect-Quorum
        Broadcast[crowd-larger];
        ]];
    Suite-Not-Found[/ not-found: Boolean] ← Critical[suite-lock, '[not-found-votes>=w]];
```

The reference produced by Suite-Create is a file reference that contains a volume suite. Expand-Suite is used by Initiate-Inquiries to convert such references into normal form.

```
Expand-Suite[ref: Suite / expanded: Normal-Suite] ← Prog[
        [f: File;  v: Volume; nv: Volume;  reps: List[Reference]];
        IF Is[ref, Normal-Suite] THEN Return[ref];
        -- ref is a result of Suite-Create
        nv ← Normalize[ref.volume, tc];
        reps ← NIL;
        FOR v IN nv.reps DO [
                f ← Create[File];  f.id ← ref.id;  f.volume ← v;
                -- if volume rep is a volume suite, make file rep a file suite
                IF Is[v, Suite] THEN Add-Type[f, Suite];
                reps ← Append[reps, f];
                ];
        Return[Create-Suite[ref.id, nv.r, nv.w, reps, nv.votes]];
        ];
```

A Close request is applied to every open representative.

```
Suite-Close[] ← Critical[suite-lock, 'Prog[ [];
        Return[Apply-Quorum[suite]];
        ]];
```

Collect-Quorum normally gathers a quorum with a specified number of votes and returns immediately to its caller. All of the representatives in the quorum are not guaranteed to be current. If a quorum of representatives have not reported their version numbers all that Collect-Quorum can do is wait for a representative to respond.

Quorum sizes are the minimum number of votes that must be collected to guarantee correct operation. However, quorums can always be expanded by adding additional representatives. In the model implementation all eligible representatives are included in every quorum.

```
Collect-Quorum[threshold: Integer/ quorum: List[Representative]] ← Prog [ [];
        DO [
                -- if the suite does not exist, return an error
                IF Suite-Not-Found[] THEN Return[Error['NotFound]];
                quorum ← Collect[threshold];
                IF Not[Null[quorum]] THEN Return[quorum];
                -- if we can't get a quorum, just wait
                Wait[crowd-larger];
                ];
        ];
Collect[threshold: Integer / quorum: List[Representative]] ← Critical[suite-lock, 'Prog [
        [x: Representative; votes: Integer];
        -- returns a quorum or NIL
        votes ← 0;
        FOR x IN suite DO votes ← votes + x.votes;
        IF votes<threshold THEN Return[NIL];
        Return[suite];
        ]];
```

Collect-Write-Quorum attempts to gather a write quorum. All of the representatives in a write quorum are guaranteed to be current. It is possible that although we have enough votes to make

up a write quorum, we do not have enough representatives that are current. Collect-Write-Quorum solves this problem by copying the contents of the suite into an available obsolete representative. It is always legal to copy the contents of the suite into an obsolete representative.

```
Collect-Write-Quorum[/ quorum: List[Representative]] ← Critical[write-lock, 'Prog [ [];
        -- if we have a quorum, return it
        IF Not[Null[write-quorum]] THEN Return[write-quorum];
        DO [
                -- try to get a write quorum
                write-quorum ← Collect-Write[];
                IF Not[Null[write-quorum]] THEN Return[write-quorum];
                -- if we have a chance, update an obsolete representative
                -- otherwise, just wait for another representative
                IF raw-votes<w THEN Wait[crowd-larger]
                        ELSE Update-Obsolete-Representative[];
                ];
        ]];
Collect-Write[/ quorum: List[Representative]] ← Critical[suite-lock, 'Prog [
        [x: Representative; votes: Integer;];
        -- try to gather a write quorum
        votes ← 0;  raw-votes ← 0;  quorum ← NIL;
        FOR x IN suite DO [
                raw-votes ← raw-votes + x.votes;
                IF x.version=current THEN [
                        -- this representative can be included
                        votes ← votes + x.votes;
                        quorum ← Append[quorum, x];
                        ];
                ];
        IF votes>w THEN Return[quorum];
        -- could not get a quorum
        Return[NIL];
        ]];
```

An obsolete representative must be updated with a consistent version of the suite. The model implementation achieves this by not allowing the suite to change while an obsolete representative is being updated. Alternatively, an obsolete representative could be updated in a separate transaction.

```
Update-Obsolete-Representative[] ← Prog [
        [rep: Representative];
        rep ← Select-Obsolete[];
        IF Not[Null[rep]] THEN [
                rep.class | Copy[suite-ref];
                Update-Done[rep];
                ];
        ];
Select-Obsolete[/ rep: Representative] ← Critical[suite-lock, 'Prog [ [];
        FOR rep IN suite DO
                IF rep.version#current THEN
                        -- We found an obsolete representative.
```

```
                    Return[rep];
        Return[NIL];
        ]];

Update-Done[rep: Representative] ← Critical[suite-lock, 'Prog[ [];
        -- mark representative as being current
        rep.version ← current;
        ]];
```

The above functions comprise the model implementation of the suite algorithm. There are many refinements that can be made to the suite we have described, and some of them are given at the end of this chapter. For example, when a suite is closed inquiry processes can optionally be stopped. The details of how this can be accomplished were not shown.

5.3 Correctness Arguments

We argue that if a representative has the highest version number it has received all of the updates to the suite. The set of representatives that we update to make version $v+1$ all have version v. The desired result follows by simple induction.

We argue that the replication algorithm preserves the properties of transactions. Representatives are stored in transactional storage, and we base our arguments on the properties transactional storage is guaranteed to exhibit.

(totality) Suite updates are transformed to representative updates. Representative updates are guaranteed totality. Thus, suite updates are guaranteed totality.

(consistency) We show that the suite algorithm preserves serial and external consistency (Section 3.1.2) by showing that it produces schedules that are equivalent to the unreplicated case's schedules. Let S be a schedule, and S' be a schedule that results when we change object O to be a suite. We wish to show that DEP(S)=DEP(S').

$\langle Ta, O, Tb\rangle \in DEP(S) \Rightarrow \langle Ta', O', Tb'\rangle \in DEP(S')$.

Without loss of generality, we only consider O. If $\langle Ta, O, Tb\rangle \in DEP(S)$ then Ta writes O and Tb reads or writes O. To write, Ta' updates a write quorum of representatives. When Tb' reads or writes it first gathers a read quorum. Read quorums and write quorums have an element in common. Thus, $\langle Ta', O', Tb'\rangle \in DEP(S')$.

$\langle Ta', O', Tb'\rangle \in DEP(S') \Rightarrow \langle Ta, O, Tb\rangle \in DEP(S)$.

Without loss of generality, we only consider O. If $\langle Ta', O', Tb'\rangle \in DEP(S')$ then Ta' writes a write quorum and Tb' reads a read quorum. Thus, Ta writes O and Tb either reads or writes O. Thus, $\langle Ta, O, Tb\rangle \in DEP(S)$.

5.4 Refinements

5.4.1 Weak Representatives

Temporary copies can be introduced into a suite by creating representatives that have no votes. Such a representative does not change the quorums of a suite, and thus can be introduced at any time. Once a suite is open, read requests can be serviced by a weak representative that is current. When a weak representative is located on a high performance storage device, it can improve the performance of the suite.

Because a weak representative has no votes, it bears no responsibility for the long term safekeeping of data. There will always be a write quorum of representatives that contain current data. Thus, weak representatives can be discarded at any time. This simplifies the concurrency and recovery requirements of weak representatives. An important result of this is that weak representatives can be kept in volatile storage.

5.4.2 Lower Degrees of Consistency

The suite algorithm provides serial consistency, but if voting rules are intentionally broken, lower degrees of consistency will result. For example, setting r to be 0 corresponds to the notion "give me the latest version you can find, but I don't care if it isn't current". Certain applications that have self-correcting characteristics, such as name lookup, can use lower degrees of consistency. If the suite algorithm is run on a file system that ensures Degree 0 or Degree 1 consistency, the algorithm will guarantee the same consistency it sees, a fact we will not prove here.

5.4.3 Representative Performance

While a suite is operating it is a simple matter to gather response time statistics for each representative. The functions that collect quorums could be modified to use this information to favor faster representatives.

5.4.4 Expressive Power of Suites

It is possible of course that a suite representative can be a suite itself. It turns out that recursion makes the suite mechanism more powerful. Consider the the set RS of read quorums and the set WS of write quorums

RS = {{1 3} {1 4} {2 3} {2 4}} WS = {{1 2} {3 4}}

It is impossible to assign weights to four representatives to achieve a sets of read and write quorums identical to RS and WS. However, let us define three suites, S1, S2, and S3. S1 will consist of the representatives {1 2}, S2 will consist of the representatives {3 4}, and S3 will consist of the representatives {S1 S2}. In S1 and S2, assign one vote to each representative, and fix r to be 1 and w to be 2. In S3, assign one vote to each representative, and fix r to be 2 and w to be 1. The read and write quorum sets of S3 are RS and WS.

5.4.5 Size of Replicated Objects

The size of an object that is replicated should be chosen to match the needs of an intended application. For example, a data base manger might choose to replicate relations or tuples. Each replicated object must be assigned a version number.

The class we described to implement suites read all of the version numbers of a suite's

representatives when it was initialized. An alternative would be to examine a read quorum on every read operation. This might be appropriate for replicating small objects that do not lend themselves to being opened.

5.4.6 Total Replacement

A suite's version number is determined by examining a read quorum. This ensures that two transactions that concurrently attempt to update a suite conflict. However, if objects are totally replaced, then we do not need to guard against concurrent updates. In this case we simply need a rule that assigns a single value to a suite. This can be accomplished by replacing version numbers with time stamps that are totally ordered. With total replacement, a current representative is found by reading the time stamps of a read quorum, and selecting a representative with the largest stamp. [Lamport 78b] discusses how to implement suitable time stamps.

5.4.7 Simultaneous Suite Updates

Our use of version numbers does not allow a suite to be updated by more than one transaction at a time. This restriction can be eliminated if objects are totally replaced (see the previous section), or if the transactional storage system provides broken read locks (Section 3.3.4).

Concurrent disjoint suite updates are not compatible to prevent different updates from selecting different write quorums. If two update transactions selected different write quorums then a representative that claimed to be current might not have received all of the updates to the suite.

If the transactional storage system provides broken read locks then disjoint suite updates can be made compatible. The replication algorithm only need ensure that a set of concurrent updates selects the same write quorum.

5.4.8 Releasing Read Locks

Every lock that a transaction holds necessitates communication at commit time to ensure that the lock is still in force (Section 3.3.5). The suite algorithm sends inquiries to all representatives in the suite to determine their status. Thus, a read lock is obtained on every available representative. An enhancement would be to release the read locks of representatives that are never included in a quorum.

5.4.9 Updating Representatives in Background

Obsolete representatives can be updated at any time. It would be possible to operate servers that examined volume suites, updating obsolete file suite representatives. This could be done when there was surplus communication capacity in the internetwork.

5.4.10 Guessing a Representative is Current

It is possible to guess that a representative is current before a read quorum has responded to an initial inquiry. If the guess is correct, then the delay to open the suite is reduced. If the the guess is wrong, then the client's transaction must be aborted.

5.4.11 Postponed Creation of File Representatives

When a file is created on a volume suite Suite-Create ensures that at least MAX[r,w] votes worth of representatives are created. File representatives could be created on the remainder of the volume

suite's representatives at a later time. After they had been created, these new representatives would be handled in the same way as obsolete representatives.

5.4.12 An Alternative for Replicated Volumes

An alternative is to implement replicated volumes with a file suite. A storage device is fundamentally just a large file, and thus a collection of storage devices could be treated as a file suite. A replicated volume could be implemented using such a file suite. This approach has merit, but updating an obsolete representative requires that an entire storage device be copied.

5.5 Related Work

The essence of the suite algorithm is also described in [Gifford 79b]. Previous algorithms for maintaining replicated data fall into two classes. Some insist that every object has a primary site which assumes responsibility for arbitrating concurrent updates. [Alsbert et al. 76] first outlined this idea. This technique is simple, but relatively inflexible. Other algorithms do not employ distinguished sites for objects, and they are more complex than primary site algorithms. SDD-1 [Rothnie et al. 77] keeps all copies of an object up to date by sending updates via a communication system that will buffer messages over machine crashes. Thomas' [Thomas 79] proposal only requires that a majority of an object's copies be updated, and includes voting.

Although we share the notion of voting, it is difficult to directly compare our algorithm with Thomas' because the two provide different services. Notably: (1) we guarantee serial and external consistency for queries (read-only transactions); (2) we do not insist that a majority of an object's copies be updated; (3) Thomas' algorithm does not employ weighted voters, which limits its flexibility; (4) Thomas' algorithm is more complex because it addresses consistency issues as well as replication issues; and (5) our structure allows for the inclusion of temporary copies.

5.6 Summary

We have described an algorithm for replicating data that offers many benefits not provided by previous solutions. The introduction of weighted voting allows suites to be synthesized with desired properties, including the presence of temporary copies. The separation of consistency considerations from replication has resulted in a conceptually simple approach which guarantees consistency in a straightforward way and is relatively easy to implement.

Chapter 6: Reconfiguration

There are many situations in which one might want to change how information is stored. For example, if a collection of files is no longer in regular use, it might be desirable to move them to a low-cost storage volume. Or perhaps we would like to replace a volume with a volume suite to improve availability.

This chapter shows how the resources that are used to implement a volume, file, or index can be changed. The first section of the chapter introduces the concepts of reconfiguration, the second section shows how these concepts can be implemented, and the final section offers some refinements to the basic algorithm.

6.1 Reconfigurable Objects

In chapter four we saw how the location of an object can be changed, and how one object can be substituted for another. For example, indirect processor references implicitly define virtual processors. Different physical processors can be bound to an indirect processor reference at different times.

However, indirect references have limitations. If we desire to create a virtual volume that is serviced by different physical volumes it is not sufficient to simply update an indirect reference. We must ensure that when a new physical volume is used it contains the same information as the old volume. This requirement holds for indexes, files, and volumes.

To solve this problem we introduce the notion of a *reconfigurable object.* A reconfigurable object can be implemented by different objects at different times, and the state of the reconfigurable object will be properly transferred when one object is substituted for another. A naive client will not know that it is using a reconfigurable object, and an object can be reconfigured while clients are accessing it.

Create-Reconfigurable[ref: Reference, index: Index, tc: TC, ring: List[Key] / rr: Reconfigurable]

> Create-Reconfigurable creates a reconfigurable object, and uses ref as the first implementation of the object. The specified index is used in the implementation of the reconfigurable object.

rc | Reconfigure[new-ref: Reference]

> Reconfigure can be applied to a class that services a reconfigurable object. Assume *RO* is the object serviced by *rc.* After Reconfigure completes, RO will be implemented by *new-ref.* *rc* will also be changed to service *new-ref.*
>
> *new-ref* can be one of two things:
>
> 1. *new-ref* can be a reference for a brand new object.
> 2. *new-ref* can describe a new suite configuration for RO. A client can create a new suite configuration with Create-Suite. Thus, Reconfigure can be used to change the voting configuration of a suite.

6.2 Algorithm

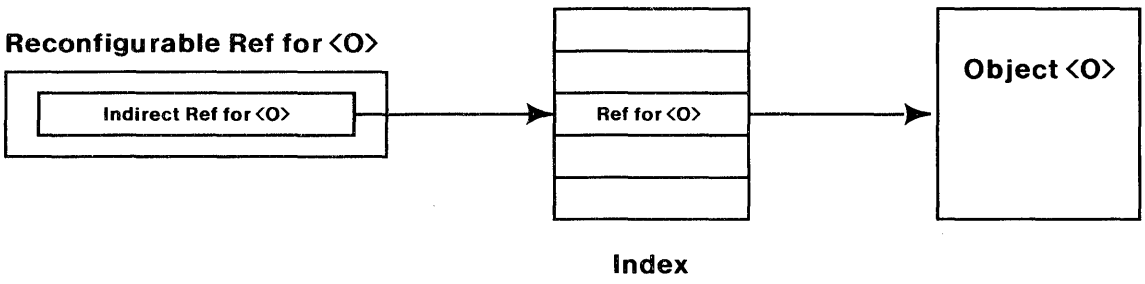
The structure of a reconfigurable reference for a file or a volume is shown in Figure 6.1. The reconfigurable reference contains an indirect reference, which points to the current implementor of the file or index.

A reconfigurable volume is a collection of files which may be reconfigured as a group. The set of files that are contained in a reconfigurable volume are kept in the reconfigurable volume's *file index*. Figure 6.2 shows the structure of a reconfigurable volume. The algorithm for volume reconfiguration is described below.

The following functions comprise the model implementation of reconfiguration.

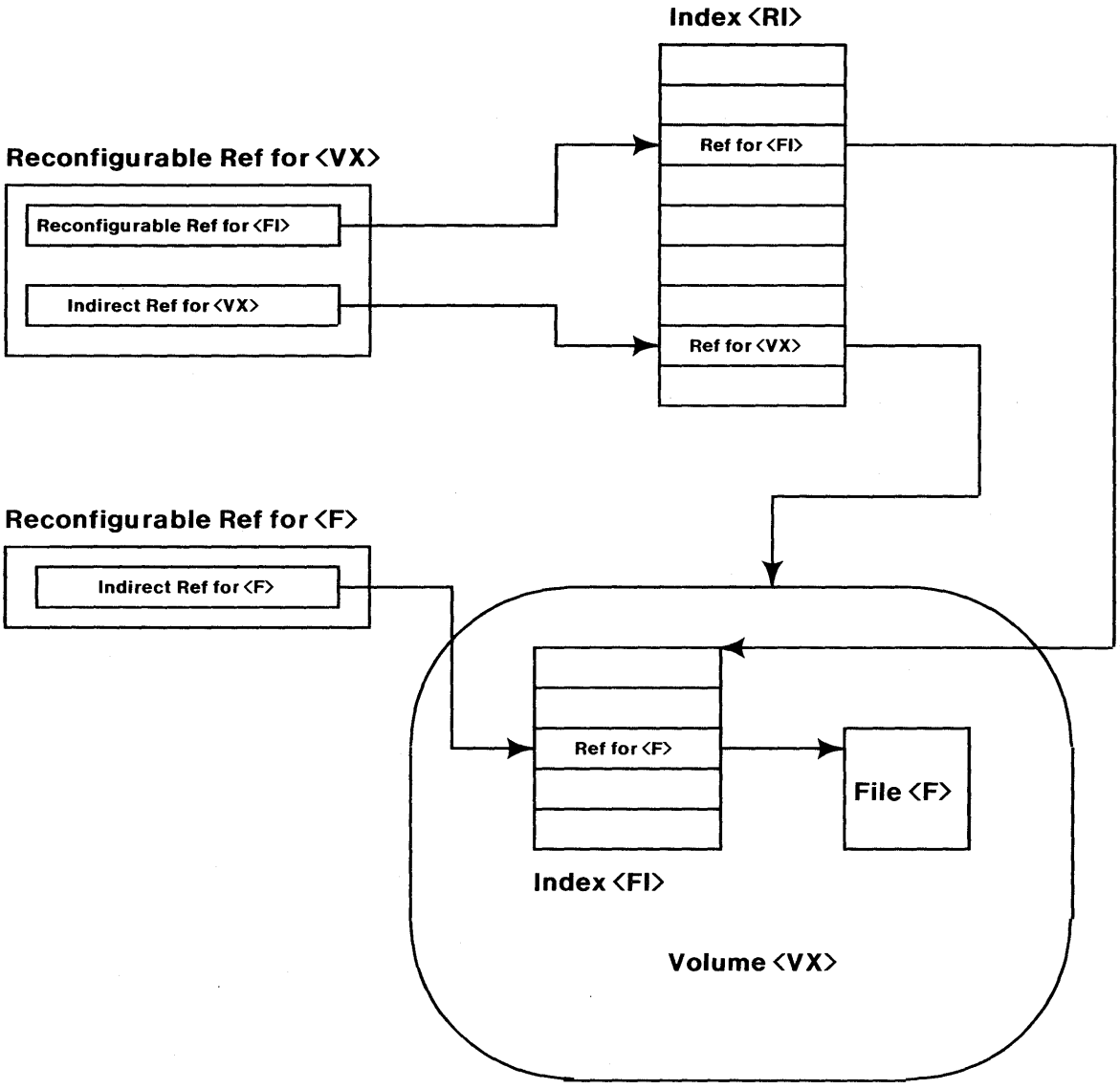
```
Reconfigurable ← Record[ref: Indirect];

Reconfigurable-Volume ← Extend[Reconfigurable, Record[file-index: Index]];

Create-Reconfigurable[ref: Reference, index: Index, tc: TC, ring: List[Key]
  / rr: Reconfigurable] ← Prog[ [tref: Reference; tclass: Class];
      IF Is[ref, Volume] THEN [
            rr ← Create[Reconfigurable-Volume];
            tclass ← Open[ref, tc, ring];
            -- create a file on volume for the file index
            tref ← tclass | Create-File[GetUniqueID[]];
            tclass | Close[];
            -- create file index
            tref ← Create-Index[tref];
            -- make the index reconfigurable
            rr.file-index ← Create-Reconfigurable[tref, index, tc, ring];
            -- create indirect for volume reference
            rr.ref ← Create-Indirect[ref, index, tc]
      ] ELSE [
            rr ← Create[Reconfigurable];
            rr.ref ← Create-Indirect[ref, index, tc];
            ];
      Return[Add-Type[rr, Major-Type[ref]]];
      ];

Open-Reconfigurable[ref: Reference, tc: TC, ring: List[Key], guards: List[Key]
  / class: Class] ← Prog[
      [rfn: Function];
      class ← Open[ref.ref, tc, ring, guards];
      IF Is[class, Error-Type] THEN Return[class];
      rfn ← IF Is[ref, Volume] THEN 'Reconfigure-Volume ELSE 'Reconfigure;
      -- Instance variables: ref, tc, ring
      Return[Create-Class[List[
            'Reconfigure,   rfn,
            'CopyReference,   'Default-Copy,
            'Create-File,   'Reconfigure-Create-File,
            ], class]];
      ];
```

Figure 6.3 shows how the reconfiguration algorithm for files and indexes works. First, the



Structure of a Reconfigurable File or Index

Figure 6.1



Structure of a Reconfigurable Volume

Figure 6.2

contents of the object is copied by its new implementor. Second, the indirect reference is changed to point at the new implementor.

```
Reconfigure[new-ref: Reference] ← Prog[
        [];
        -- for files and indexes only
        ic ← Open[new-ref, tc, ring];
        ic | Copy[ref];
        ic | Close[];
        superclass | ChangeReference[new-ref];
        ];
```

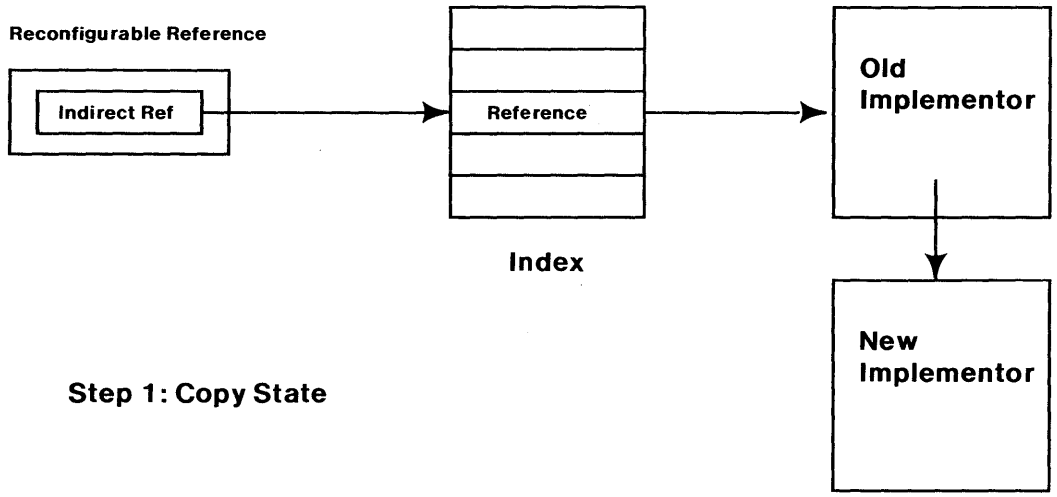
Whenever a file is created on a reconfigurable volume, it is made reconfigurable. The volume's file index is used to implement the necessary indirection.

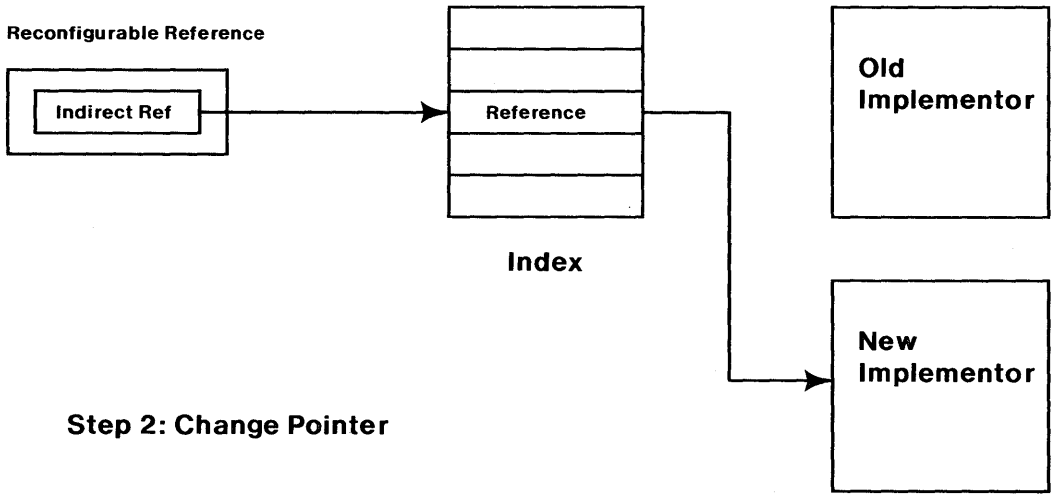
```
Reconfigure-Create-File[] ← Prog[
        [file-ref: File-Reference];
        file-ref ← Apply[superclass, request];
        IF Is[file-ref, Error-Type] THEN Return[file-ref];
        Return[Create-Reconfigurable[file-ref, ref.file-index, tc]];
        ];
```

As shown in Figures 6.4 through 6.6, reconfiguring a volume occurs in three steps:

1. All of the files on the volume are moved to their new home. This is accomplished by moving the files that are listed in the file index one by one.
2. The file index is moved to the new volume.
3. The indirect pointer to the current implementor of the reconfigurable volume is switched to point at the new volume.

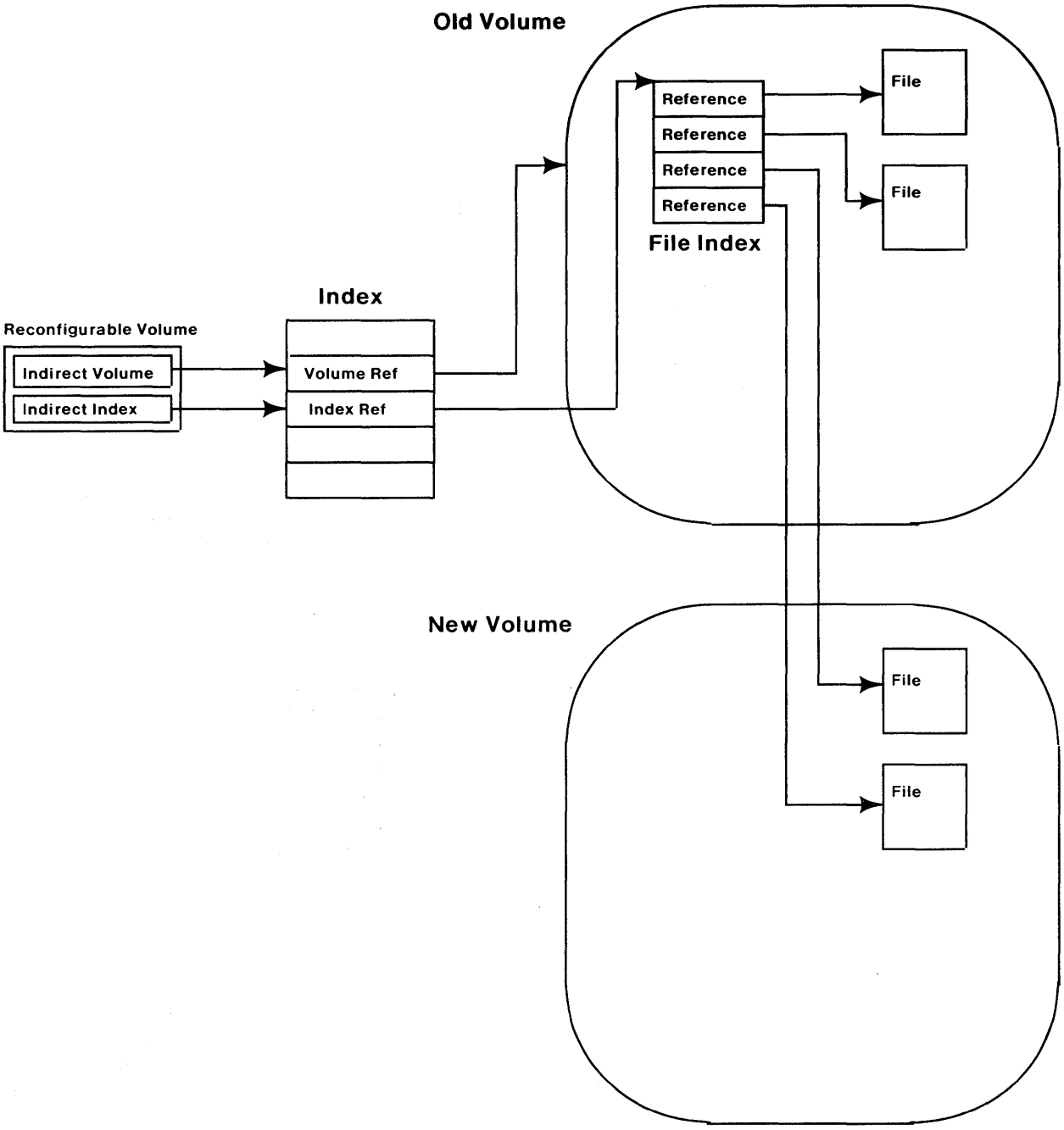
Contention for a reconfigurable volume may cause a transaction that is performing a reconfiguration to abort. Step 1 can be split up into many smaller transactions, each of which would reconfigure a single file.





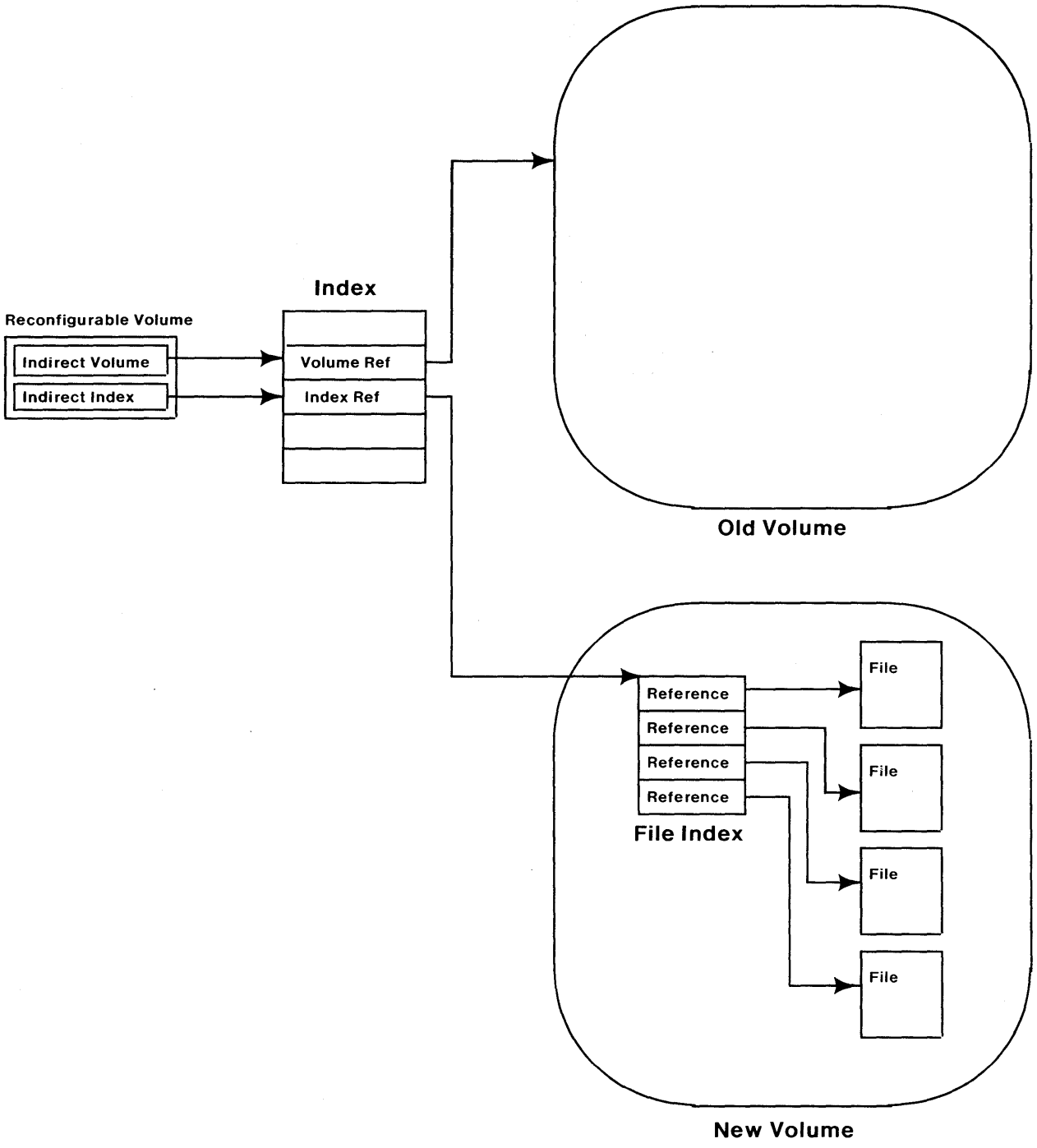
File and Index Reconfiguration Algorithm

Figure 6.3



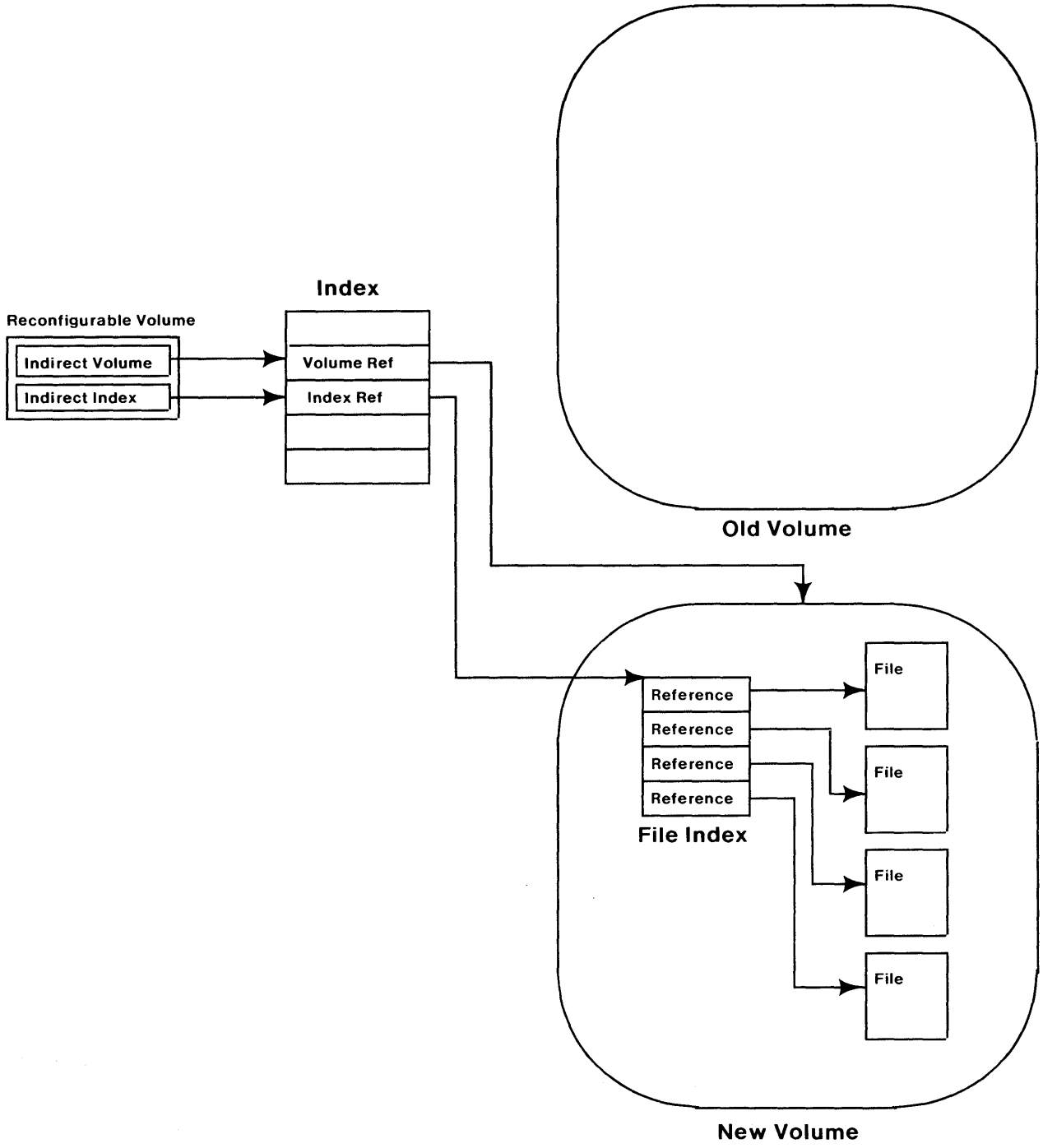
Step 1 of Volume Reconfiguration: Move Files

Figure 6.4



Step 2 of Volume Reconfiguration: Move File Index

Figure 6.5



Step 3 of Volume Reconfiguration: Change Volume Pointer

Figure 6.6

```
Reconfigure-Volume[new-volume: Volume] ← Prog[
        [old-home: Reference; new-home: Reference; entry: Entry; vc: Volume-Class;
        ic: Index-Class];
        -- reconfigure all of the files on the volume
        ic ← Open[ref.file-index, tc, ring];
        vc ← Open[new-volume, tc, ring];
        entry ← ic | Enumerate[NIL];
        -- move all of the files
        WHILE Not[Null[entry]] DO [
                -- move one file: first construct a reference
                rfr ← Create[Reconfigurable];
                rfr.ref ← Create[Indirect];
                rfr.ref.index ← ref.file-index;
                rfr.ref.indirect-id ← entry.entry-name;
                -- open current implementor of file
                fc ← Open[rfr, tc, ring];
                -- create new implementor of file
                new-home ← vc | Create-File[fc | GetID[], T];
                -- move file
                fc | Reconfigure[new-home];
                fc | Close[];
                entry ← ic | Enumerate[entry];
                ];
        -- now move the file index
        new-home ← vc | Create-File[ic | GetID[], T];
        new-home ← Create-Index[new-home];
        ic | Reconfigure[new-home];
        -- change the volume's configuration
        superclass | ChangeReference[new-volume];
        -- all done.
        ic | Close[];
        vc | Close[];
        ];
```

6.3 Refinements

6.3.1 Reducing Data Movement

Copy could be modified to return if its source and destination were already identical. Copy could accomplish this by checking their unique identifiers and version numbers.

This could significantly reduce unnecessary data movement. Consider the case of a file suite that is reconfigured to change its vote assignments. It may be that no representatives need to be updated. In this case, with the proposed improvement to Copy, the Reconfigure operation would not cause any data movement.

6.3.2 Eliminating the File Index

The file index for a reconfigurable volume could be eliminated by storing a file's index entry in its first few pages. When such a file was opened, it would either contain a pointer to itself or to its new location. The reason that we did not use this scheme is that if a file that is part of such an

indirect chain was unavailable, it would be impossible to find the new home of a file. By using an index we avoided this problem at some expense.

6.4 Summary

Reconfiguration was accomplished with indirection and transfer of state at the time an object is reconfigured. All of the information containing objects we have defined - volumes, files, and indexes - can be made dynamically reconfigurable.

Exercise

1. Discuss how a volume reconfiguration could use a transaction for each file or index that had to be moved.

Chapter 7: Protection

Until this point we have assumed that users are perfectly trustworthy. This chapter relaxes this assumption by describing a protection mechanism that can be used to protect client information. The mechanism can be used directly for the protection of small objects such as data base entries, and it can be used to implement popular protection policies for larger objects.

Protection can be considered to consist of four distinct components: *secrecy* (ensuring that information is only disclosed to authorized users), *authentication* (ensuring that information is not forged), *integrity* (ensuring that information is not destroyed), and *availability* (ensuring that access to information can not be maliciously interrupted).

This chapter describes a new protection mechanism called *cryptographic sealing* that provides primitives for secrecy and authentication. The mechanism is enforced with a synthesis of conventional cryptography, public-key cryptography, and a threshold scheme.

The new mechanism is based on the idea of *sealing* an object with a *key*. *Sealed objects* are self-authenticating, and in the absence of an appropriate set of keys, only provide information about the size of their contents. Thus, keys are the basic unit of secrecy and authentication in the mechanism. New keys can be freely created at any time, and keys can also be derived from existing keys with operators that include *Key-Or* and *Key-And*. These operators allow protection structures to be established that allow any member of a set of keys to unseal an object, that require every member of a set of keys to unseal an object, or any combination of these extremes. This flexibility allows cryptographic sealing to implement common protection mechansims such as capabilities, access control lists, and information flow control.

Objects and sealed objects are simply arrays of bytes. In order to update the value of a sealed object it is necessary to unseal it, change its value, and reseal it.

Clients must operate in a secure environment so they can safely manipulate unsealed objects. A simple way of providing such an environment would be for each client to execute on a separate physical processor. For the purposes of this discussion we will assume that every client executes on a secure processor that is protected from every other client (see Section 2.1.1).

Our description of the protection mechanism and its applications is organized into six sections. The first section describes some preliminaries, including the general framework of the mechanism and the cryptographic methods that we use. The second section covers cryptographic sealing. The third section shows how cryptographic sealing can be used to implement capabilities, access control lists, information flow control, secure processors, and revocation. The fourth section examines some practical considerations. The fifth section is a comparative analysis of cryptographic sealing and traditional protection mechanisms. The chapter ends with a brief conclusion.

7.1 Preliminaries

7.1.1 Framework

In order to compare the present work with previous protection systems [Saltzer and Schroeder 79] we need to provide an appropriate framework. We will call a protection mechanism *active* if it is placed between a client and protected information, as shown in Figure 7.1. An active protection mechanism serves to inhibit unauthorized client requests to storage. If a client could bypass an active protection mechanism, it could gain unauthorized access to protected information. Thus, active protection mechanisms depend on a *security envelope* that a client can not penetrate. As shown in the figure, system administrators and operators are inside of this envelope, because they have direct physical access to the system.

The protection mechanism this chapter introduces is the first example of a general purpose *passive* protection mechanism. It is passive because there are no restrictions placed on a client's access to storage. However, a client is only able to decipher information that it is authorized to see. As shown in Figure 7.2, a passive protection system operates as part of the client.

A passive protection system can only address the secrecy and authentication aspects of computer security. Section 7.3 describes how an active protection mechanism can be used to supplement a passive system to provide integrity and availability. When passive and active systems are used together in this manner a failure of the active protection system only effects integrity and availability. Secrecy and authentication are still guaranteed by the passive protection system. In a similar way, if the ability to revoke privileges is desired, an active protection system would be used as described in Section 7.3.4.

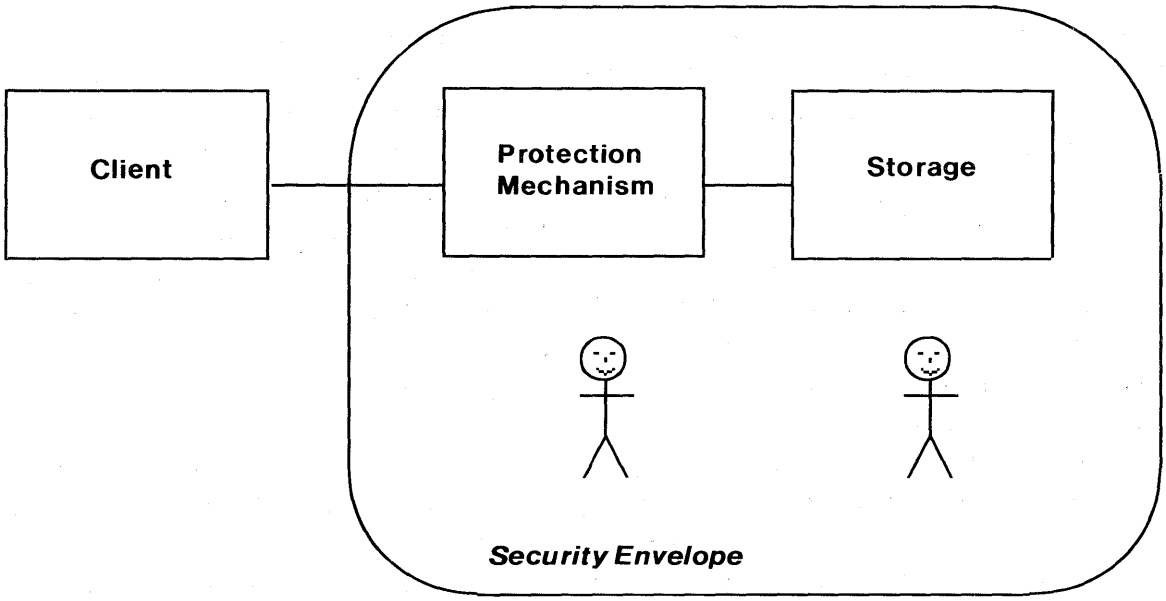
Some of the ideas in this chapter have appeared in other forms. Morris [Morris 73] discussed the concept of sealing objects, but he did not present a way to create general protection structures, and his mechanism was not enforced by cryptography. Chaum and Fabry [Chaum and Fabry 78], Lindsay and Gligor [Lindsay and Gligor 78], and Needham [Needham 79] independently observed that cryptography can be used to authenticate objects such as capabilities. Gudes [Gudes 80] described a way to use cryptography to implement a form of access control lists without groups or indirect keys.

7.1.2 Environment

We describe the building blocks of the protection mechanism in detail because with a thorough knowledge of these facilities the reader will find it easier to understand the sections that follow. The three facilities that the protection mechanism uses are cryptography, a threshold scheme, and checksums. It is difficult to motivate all of these facilities in the abstract, and thus we ask the reader to be patient until the following section where it will become clear why they were introduced.

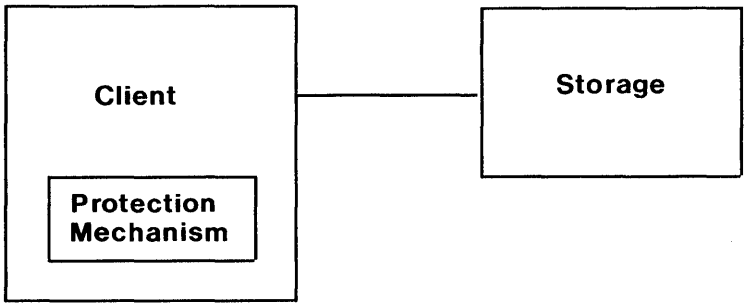
7.1.2.1 Cryptography

Cryptography is used to *encrypt* information to be protected, also called *cleartext*, into *ciphertext*. Encryption is a transformation from the space of possible cleartexts to the space of possible ciphertexts. The transformation selected depends on the key supplied to the encryption function. To *decrypt*, or recover the cleartext of a ciphertext, requires a specific key so the



An Active Protection Mechanism

Figure 7.1



A Passive Protection Mechanism

Figure 7.2

encryption transformation can be inverted.

A cryptographic system is called *perfectly secure* if all that can be discerned from a ciphertext is that the ciphertext exists, and that the corresponding cleartext is not longer than the ciphertext. Perfect security can only be achieved when the entropy of a key is at least as great as the entropy of information encrypted [Shannon 49]. Thus, perfectly secure systems are usually not practical because a key must be at least as long as the cleartext it is used to encrypt.

A cryptographic system is called *computationally secure* if even when enough information is theoretically available to break the system, the amount of computation required to do so is unreasonable. Unfortunately, proving facts about the computational complexity of many interesting algorithms is beyond the reach of current theory. Thus, most practical cryptosystems can not be proven to be computationally secure. In practice, the best that can be done is to invest a substantial amount of effort in trying to think of a way to break a system; if the system survives such an attack, then it is considered to be secure for practical purposes. As unsatisfying as this may be, it is the current state of the art.

The goal of a cryptanalytic attack is to find the key that was used to encrypt a ciphertext. Attacks are classified by the information an intruder has to aid him. It is assumed that an intruder knows the workings of the cryptosystem he is attacking. As the name implies, in a *ciphertext-only attack* an intruder only has ciphertext. In a *known-cleartext attack* an intruder has both the ciphertext and the cleartext that was used to generate it. In a *chosen-cleartext attack* an intruder can choose text to be encrypted and observe the resulting ciphertext. Experience has shown that a cryptosystem that is known to be vulnerable to a chosen-cleartext attack should not be used.

Two generic types of cryptosystems are used to implement cryptographic sealing. As we introduce each cryptosystem we will provide enough background so the reader can understand its properties. For more information about cryptography see Diffie and Hellman [Diffie and Hellman 79].

The framework of the protection system is very general, and it is designed to accommodate new cryptosystems as they are discovered. The interfaces to the cryptosystems are described in general terms, allowing stronger implementations of the systems to be adopted as they become available.

Conventional Cryptography

In a *conventional cryptosystem* a cleartext is encrypted with a key, and the same key is used to decrypt the resulting ciphertext. The following functions implement conventional cryptography.

Create-Conventional-Key[/key: Byte-Array]

> Create-Conventional-Key returns a random key that can be used in conjunction with the conventional encrypt and decrypt functions. All of our key creation functions are based on a true random bit generator.

Conventional-Encrypt[clear: Byte-Array, key: Byte-Array / cipher: Byte-Array]

> Conventional-Encrypt encrypts *clear* with *key*, producing *cipher*.

Conventional-Decrypt[cipher: Byte-Array, key: Byte-Array / clear: Byte-Array]

> Conventional-Decrypt decrypts *cipher* with *key*, producing *clear*. If the key

specified was not used to encrypt *cipher*, the value of *clear* is undefined.

Public-Key Cryptography

In a *public-key cryptosystem* the key generation function returns a pair of keys. Call the keys of such a pair *keya* and *keyb*. Only *keyb* can decrypt ciphertext enciphered with *keya*, and only *keya* can decrypt ciphertext enciphered with *keyb*. This is in contrast to a conventional cryptosystem, where a single key is used for both encryption and decryption.

Public-key cryptosystems were first suggested in the open literature by Diffie and Hellman [Diffie and Hellman 76]. The name public-key resulted from the observation that with such a system certain keys could be made public, solving in part the problem of key distribution. Public-key cryptosystems have also been called asymmetric cryptosystems.

Rivest, Shamir, and Adleman [Rivest et al. 78] described the first practical implementation of a public-key system. The only cryptanalytic approaches currently known for breaking their scheme are at least as computationally complex as factoring extremely large numbers.

The exact semantics we chose for our definition of public-key cryptography were motivated by the Rivest, Shamir, and Adleman proposal. Some public-key cryptosystems allow ciphertext enciphered with *keya* to be decrypted with *keyb*, but not the converse. In this case we will assume that two key pairs from such a system are used to implement one of our public-key pairs to provide the appropriate semantics.

The following functions implement public-key cryptography.

PK-Pair ← Record[keya: Byte-Array, keyb: Byte-Array];

Create-PK-Pair[/pair: PK-Pair]

> *Create-PK-Pair* computes a pair of public keys. The keys are random, in the sense that they are a function of a true random number.

PK-Encrypt[clear: Byte-Array, key: Byte-Array / cipher: Byte-Array]

> *PK-Encrypt* encrypts *clear* with one of the members of a key pair, and it returns *cipher*.

PK-Decrypt[cipher: Byte-Array, key: Byte-Array / clear: Byte-Array]

> *PK-Decrypt* decrypts *cipher* with *key*, producing *clear*. If the key specified is not correct, the value of *clear* is undefined.

7.1.2.2 Threshold Scheme

A *threshold scheme* allows a datum *D* to be divided into *n* pieces, such that any *k* pieces are sufficient to reconstruct *D* but complete knowledge of any *k-1* pieces reveals no information about *D* [Blakley 79, Shamir 79]. A practical implementation of this system was first demonstrated by [Shamir 79].

The following functions implement a threshold scheme.

Threshold-Pieces ← Record[pieces: List];

Threshold-Split[clear: Byte-Array, n: Integer, k: Integer / piece-list: Threshold-Pieces]

Threshold-Split takes *clear* and logically splits it into *n* pieces, any *k* of which are sufficient to recover *clear*. These pieces are returned as an *n* element list.

Threshold-Recover[piece-list: Threshold-Pieces / clear: Byte-Array]

Threshold-Recover takes a list of pieces, and if *k* pieces are available, it will return the original value of *clear*. Elements on the input list that are not threshold pieces are ignored. If *k* pieces are not available, *Threshold-Recover* returns *Error['Failed]*.

7.1.2.3 Checksums

A *checksum function* maps arbitrary input values into a comparatively small set of output values, such that independent input values have a small probability of being mapped into the same output value. If a *c* bit checksum is implemented by a cyclic code, it can be proved that the fraction of independent values that have the same checksum is 2^{-c} [Peterson and Weldon 72, p. 229]. Conventional encryption can also be used to create checksums. One way of creating an encryption based checksum is to use a cipher block chain technique [NBS 80].

Add-Checksum[x: Any / cx: Checksummed-Object]

Add-Checksum returns a copy of *x* and *x*'s checksum.

Check-Checksum[cx: Checksummed-Object / x: Any]

Check-Checksum checks the checksum of *cx*, and if the checksum is correct, it returns the value of *x* it holds. If the checksum of *cx* is incorrect, *Check-Checksum* returns *Error['Failed]*.

7.2 Cryptographic Sealing

Our basic protection mechanism, cryptographic sealing, is described in three stages. First, we provide a model of what the mechanism does. Second, we show how the mechanism works. Third, we discuss the extent to which the mechanism can be trusted.

In the sections that follow, we extend the notion of keys. These extended keys are similar enough to cryptographic keys that we did not coin a new term, but the reader should be aware that they are not precisely the same.

7.2.1 Model

As we have mentioned, keys are the basic unit of secrecy and authentication in the protection mechanism. There are two ways to generate a key. It is possible to generate a brand new random key that does not depend on any other keys; such a key is called a *base key*. It is also possible to generate a key that is a function of existing keys; such a key is called a *derived key*. Derived keys are used to implement protection structures that can not be realized with base keys alone.

The keys required to unseal an object depend on the structure of the key used to seal the object. For example, to unseal an object that was sealed with *Key-Or[ka, Key-And[kb, kc]]* one needs either *ka* or both *kb* and *kc*. To be as explicit as possible, we introduce the *unseals* relation between a set of keys and a key. As we shall see in a moment, if a set of keys *S* unseals key *k*,

then S can be used to unseal any object that has been sealed with k. If key set S unseals k we will write $S >> k$. If $S >> k$ we also say that k *is unsealed by* S.

Seal and *Unseal* are the primitive functions of the protection mechanism. *Seal* seals an object with a key, and *Unseal* reverses the sealing process. The type *Key* is used below to denote either a base or a derived key.

Seal[x: Any, k: Key / sx: Sealed-Object]

> *Seal* seals x with k, returning a sealed object. *Seal* operates by value; x and k are not modified. *Seal[x, NIL]* is x.

Unseal[sx: Sealed-Object, ks: List[Key], tc: TC / x: Any]

> *Unseal* unseals *sx* with the set of keys contained in the key set *ks*. If *Unseal* is successful, it returns x (the original input value to *Seal*). Otherwise *Unseal* returns *Error['Failed]*. Unseal does not modify *sx*.

The protection mechanism provides two properties. The *secrecy property* states that a sealed object is useless to someone who does not have a set of keys that unseals the key that was used to seal the object. The *authentication property* states that *Unseal* will only return values that were in fact properly sealed.

> **Secrecy.** x can be recovered from *Seal[x, k]* with a set of keys S if and only if $S >> k$.
>
> **Authentication.** If x was not sealed with a key that is unsealed by S then the result of *Unseal[x, S]* will be *Error['Failed]*.

The following functions create keys, and completely define the unseals relation.

Create-Base-Key[/ k: Key]

> *Create-Base-Key* creates a new regular base key. Regular base keys are the simplest kind of keys. To unseal *Seal[x, k]* requires k. In other words, a key set unseals k if and only if it contains k.
>
> $$(S >> k) \equiv (k \in S)$$

Key-Pair ← Record[keya: Key, keyb: Key];

Create-Key-Pair[/ kp: Key-Pair]

> *Create-Key-Pair* creates a pair of base keys. These keys are related to each other in the following way. To unseal *Seal[x, keya]* requires *keyb*, and to unseal *Seal[x, keyb]* requires *keya*. In other words, a key set unseals *keya* if and only if it contains *keyb*, and a key set unseals *keyb* if and only if it contains *keya*.
>
> $$(S >> keya) \equiv (keyb \in S)$$
> $$(S >> keyb) \equiv (keya \in S)$$

Key-And[ka: Key, kb: Key / dk: Key]

> *Key-And* creates a derived key that is the logical and of *ka* and *kb*. To unseal *Seal[x, dk]* requires either *dk* or a set of keys that will unseal both *ka* and *kb*. That is, a key set

unseals *dk* if and only if it unseals both *ka* and *kb* or it includes *dk*.

$$(S \gg dk) \equiv ((S \gg ka) \wedge (S \gg kb)) \vee (dk \in S)$$

Key-Or[ka: Key, kb: Key / dk: Key]

Key-Or creates a derived key that is the logical or of *ka* and *kb*. To unseal *Seal[x, dk]* requires either *dk* or a set of keys that unseals *ka* or *kb*. That is, a key set unseals *dk* if and only if it unseals *ka* or *kb* or it includes *dk*.

$$(S \gg dk) \equiv (S \gg ka) \vee (S \gg kb) \vee (dk \in S)$$

Key-Quorum[q: Integer, key-list: List[Key] / dk: Key]

Key-Quorum creates a derived key. *key-list* is a list of an arbitrary number of keys. To unseal *Seal[x, dk]* requires either *dk* or a set of keys that unseals *q* distinct keys from *key-list*. In other words, a set of keys unseals *dk* if and only if it includes *dk* or if it unseals *q* distinct keys from *key-list*. Note that *Key-Quorum* can be used to implement *Key-And* and *Key-Or* (with $q=2$ and $q=1$ respectively), but as we shall see, the implementations of *Key-And* and *Key-Or* are more efficient than *Key-Quorum*. For some *q* combination $k_1, ..., k_q$ drawn from *key-list*;

$$(S \gg dk) \equiv (\wedge_{1 \leq i \leq q} S \gg k_i) \vee (dk \in S)$$

Submaster[k: Key / dk: Key]

Submaster creates a derived key. To unseal *Seal[x, dk]* requires either *dk* or a set of keys that unseals *k*. In other words, a key set unseals *dk* if it unseals *k* or if it includes *dk*.

$$(S \gg dk) \equiv (S \gg k) \vee (dk \in S)$$

Seal-Only[k: Key / dk: Key]

Seal-Only creates a derived key. To unseal *Seal[x, dk]* requires a set of keys that unseals *k*. In other words, a key set unseals *dk* if and only if it unseals *k*. A key set that includes *dk* is not sufficient to unseal *dk*.

$$(S \gg dk) \equiv (S \gg k)$$

Create-Indirect-Key[k: Key, tc: TC / ik: Indirect-Key]

Create-Indirect-Key creates an indirect key. To unseal *Seal[x, ik]* either requires *ik* or a set of keys that unseals *k*. In other words, a key set will unseal *ik* if it contains *ik* or if it unseals *k*. Once an indirect key has been created, it can be changed with *Change-Indirect-Key*. Because indirect keys can be altered, the following statement is an implication.

$$(S \gg k) \vee (ik \in S) \rightarrow (S \gg ik)$$

Change-Indirect-Key[ik: Indirect-Key, nk: Key, tc: TC]

Change-Indirect-Key changes an indirect key. After Change-Indirect-Key has been performed, to unseal an object that has been sealed with *ik* either requires *ik* or a set of keys that unseals *nk*. In other words, *ik* is changed such that if a key set unseals *nk* it now

also unseals *ik*. *Change-Indirect-Key* does not provide revocation as discussed in Section 7.3.4.

(S >> nk) → (S >> ik)

Let us consider an example to make the idea of an indirect key clear. Imagine that the following occurs:

kevin ← Create-Base-Key[];
ik ← Create-Indirect-Key[kevin];
sx ← Seal[x, ik];

At this point *sx* can be unsealed with either *ik* or *kevin*. The indirect key is then updated:

harry ← Create-Base-Key[];
Change-Indirect-Key[ik, Key-Or[kevin, harry]];

After the Change-Indirect-Key occurs, *ik*, *harry*, or *kevin* are required to unseal *sx*.

If *S >> k1* and *S >> k2*, then *Unseal[Seal[Seal[x, k1], k2], S]* will be *Seal[x, k1]*. In order to recover an object that may have been sealed many times *RUnseal* can be used.

RUnseal[sx: Sealed-Object, ks: List[Key], tc: TC / x: Any]

RUnseal applies *Unseal* as many times as necessary until its result is not sealed. *RUnseal* always applies *Unseal* at least once.

RUnseal-List[sl: List[Sealed-Object], ks: List[Key], tc: TC / lx: List[Any]]

RUnseal-List is a function that when passed two lists [*a1* ... *an*] [*k1* ... *kn*] returns [*RUnseal[a1, k1, tc]* ... *RUnseal[an, kn, tc]*]

Seal-List[lista: List[Any], listb: List[Key] / slist: List[Sealed]]

Seal-List is a function that when passed two lists [*a1* ... *an*] [*k1* ... *kn*] returns [*Seal[a1, k1]* ... *Seal[an, kn]*].

NA-Unseal[sx: Sealed[Any], kl: List[Key], tc: TC / x: Any]

NA-Unseal unseals *sx* with RUnseal if it is sealed, otherwise it returns *sx*. Thus, NA-Unseal does not provide authentication.

NA-Unseal-List[sx: Sealed[Any], kl: List[Key], tc: TC / x: Any]

The same as RUnseal-List, except that is uses NA-Unseal.

7.2.2 Basic Algorithm

We present the basic algorithm by first outlining the principles of the mechanism and then presenting detailed descriptions of the major functions.

The protection mechanism operates by arranging that precisely enough information is included in a sealed object to allow an appropriate set of keys to unseal it. We will first consider base keys and then show how this is arranged for derived keys.

Base keys are directly implemented by cryptography. If an object is sealed with a base key,

then *Seal* encrypts the object with the key. To provide the authentication property, a checksum is added to the object before it is encrypted. *Unseal* uses the checksum to tell if it has a legitimate sealed object. For example, assume that *kb* is a base key. *Seal[x, kb]* is transformed to:

Encrypt[Add-Checksum[x], kb].

Base keys are assigned unique identifiers which are included in the result of an *Encrypt* operation. Thus, it is easy for *Unseal* to determine which key can be used to recover the contents of a sealed object.

A derived key consists of two fields: *key* and *opener*. When an object is sealed with a derived key *dk*, it is sealed with *dk.key*, and *dk.opener* is carried along in the sealed object. *Seal[x, dk]* is transformed to:

[Seal[x, dk.key], dk.opener].

The central idea is that the opener of a derived key is carefully constructed so that if $S >> dk$, then it is possible to compute a key that unseals *dk.key* from *dk.opener* and *S*.

Figure 7.3 gives several examples of sealed objects. The first example shows what happens when an object is sealed with a base key. The object is encrypted (as suggested by the heavy box), and marked with the key that can be used to decrypt the object. Although it is not shown in the figure, the object in the box includes its checksum for authentication. The next three examples show how derived keys work. It should be clear from the illustration how the opener expresses the protection structure that corresponds to its *Seal* statement.

The following functions implement the protection mechanism. Figure 7.4 is a diagram of the dependency relationships between the functions, which the reader may find helpful.

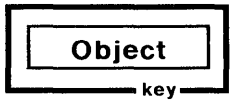
Simple base keys are implemented by conventional cryptography. A simple base key consists of a unique identifier and an encryption key.

```
Simple-Key ← Record[id: UniqueID, key: Byte-Array];

Create-Base-Key[/k: Key] ← Prog[ [];
        k ← Create[Simple-Key];
        k.key ← Create-Conventional-Key[];
        k.id ← GetUniqueID[];
        Return[k];
        ];
```

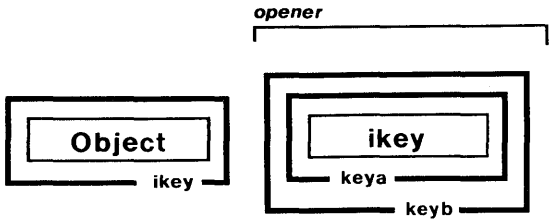
Key pairs are implemented by public-key cryptography. Like a conventional key, a *Key-Pair-Half* has fields for an identifier and a key. A *Key-Pair-Half* also includes the identifier of the key that will decrypt it.

```
Key-Pair-Half ← Record[id: UniqueID, decryptedBy: UniqueID, key: Byte-Array];

Create-Key-Pair[/ kp: Key-Pair] ← Prog[ [pkp: PK-Pair];
        kp ← Create[Key-Pair];
        kp.keya ← Create[Key-Pair-Half];
        kp.keyb ← Create[Key-Pair-Half];
        pkp ← Create-PK-Pair[];
        -- set up keya
        kp.keya.id ← GetUniqueID[];
        kp.keya.decryptedBy ← GetUniqueID[];
```

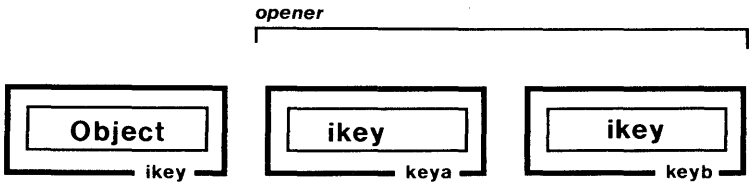
Seal[Object, key]



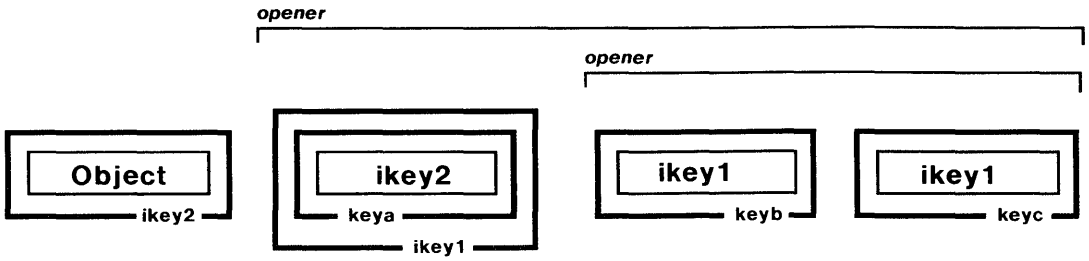
Seal[Object, Key-And[keya, keyb]]



Seal[Object, Key-Or[keya, keyb]]

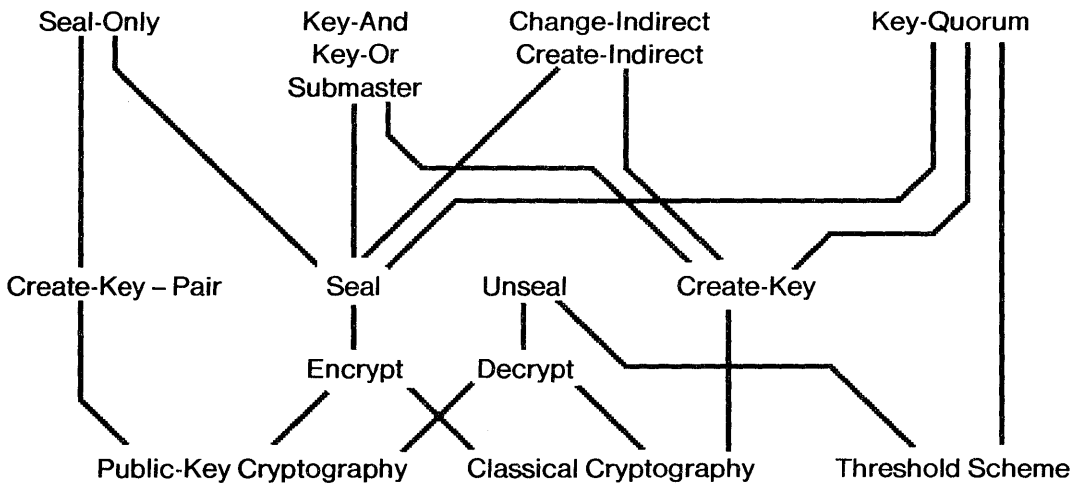


Seal[Object, Key-And[keya, Key-Or[keyb, keyc]]]



Example Sealed Objects

Figure 7.3



Protection System Internal Structure

Figure 7.4

```
        kp.keya.key ← pkp.keya;
        -- set up keyb
        kp.keyb.id ← kp.keya.decryptedBy;
        kp.keyb.decryptedBy ← kp.keya.id;
        kp.keyb.key ← pkp.keyb;
        Return[kp];
        ];
```

Before we discuss *Seal* and *Unseal*, let us introduce two cryptographic functions that work with base keys. The *Encrypt* function encrypts an object with a base key, and includes in its result the unique identifier of the key that will decrypt the resulting ciphertext. The *Decrypt* function takes an object that has been encrypted and attempts to decrypt it with the aid of a set of keys.

```
Ciphertext ← Record[decryptedBy: UniqueID, text: Byte-Array];

Encrypt[clear: Any, k: Key / cipher: Ciphertext] ← Prog[ [];
        cipher ← Create[Ciphertext];
        -- encrypt clear with k, producing cipher
        IF Is[k, Key-Pair-Half] THEN [
                cipher.decryptedBy ← k.decryptedBy;
                cipher.text ← PK-Encrypt[Encode[clear], k.key];
                ]
        ELSE [
                cipher.decryptedBy ← k.id;
                cipher.text ← Conventional-Encrypt[Encode[clear], k.key];
                ];
        Return[cipher];
        ];

Decrypt[cipher: Ciphertext, ks: List[Key] / clear: Any] ← Prog[
        [k: Key, rk: Key];
        -- decrypt cipher with one of the keys in ks
        clear ← Error['Failed];
        FOR k IN ks DO [
                -- If a derived key, get the base key hidden inside.
                rk ← IF Is[k, Derived-Key] THEN k.key ELSE k;
                -- see if we have a match
                IF Is[rk, Simple-Key] THEN [
                        IF cipher.decryptedBy = rk.id THEN
                          clear ← Decode[Conventional-Decrypt[cipher.text, rk.key]];
                        ];
                IF Is[rk, Key-Pair-Half] THEN [
                        IF cipher.decryptedBy = rk.id THEN
                          clear ← Decode[PK-Decrypt[cipher.text, rk.key]];
                        ];
                ];
        Return[clear];
        ];
```

Seal works in the following way. It first tests the type of key that it has been passed. If it is a base key then *Encrypt* is used to encrypt the object and a checksum of the object. The checksum is

used by *Unseal* to authenticate the sealed object. If *Seal* is passed a derived key it uses the base key contained in the derived key to seal the object, and it includes the opener of the derived key in its result.

```
Derived-Key ← Record[key: Key, opener: List[Sealed-Object]];

Sealed-Object ← Record[cipher: Any, opener: Any];

Seal[x: Any, k: Key / sealed-x: Sealed-Object] ← Prog[ [];
        IF Null[k] THEN Return[x];
        sealed-x ← Create[Sealed-Object];
        IF Is[k, Derived-Key] THEN [
                -- Include the derived key's opener in the sealed object.
                sealed-x.cipher ← Seal[x, k.key];
                sealed-x.opener ← k.opener;
                Return[sealed-x];
                ];
        -- Add a checksum for authentication.
        sealed-x.cipher ← Encrypt[Add-Checksum[x], k];
        sealed-x.opener ← NIL;
        Return[sealed-x];
        ];
```

Unseal reverses the sealing process. *Unseal* first checks to see what type of key was used to seal an object. If the object was sealed with a base key, *Unseal* uses *Decrypt* to recover the object's contents. The result of *Decrypt* is authenticated by ensuring that the object's checksum is correct. If the object was sealed with a derived key, the opener is unsealed, and the resulting key set is used to unseal the object.

```
Unseal[sealed-x: Sealed-Object, ks: List[Key], tc: TC / x: Any] ← Prog[
        [opener: List[Key]; tp: Threshold-Pieces];
        -- Ensure that the object was in fact sealed.
        IF Not[Is[sealed-x, Sealed-Object]] THEN Return[Error['Failed]];
        IF Null[sealed-x.opener] THEN [
                -- object was sealed with a base key
                Return[Check-Checksum[Decrypt[sealed-x.cipher, ks]]];
                ];
        -- The object was sealed with a derived key.
        opener ← Normalize[sealed-x.opener, tc];
        IF Is[opener, Threshold-Pieces] THEN [
                -- The opener is a list of sealed key pieces.
                tp ← Create[Threshold-Pieces];
                tp.pieces ← RUnseal-List[opener.pieces, ks, tc];
                opener ← List[Decode[Threshold-Recover[tp]]];
                ]
        ELSE
                -- The opener is a list of sealed keys.
                opener ← RUnseal-List[opener, ks, tc];
        Return[Unseal[sealed-x.cipher, Append[opener, ks], tc]];
        ];
```

The derived key *dk* that *Key-And* creates is simple in concept. The opener of *Key-And* can be unsealed with a key set *S* only if *S* >> *ka* and *S* >> *kb*. *ka* and *kb* must not be a key pair. This is because with the implementation of public-key cryptography proposed by [Rivest et al. 78] encrypting with one and then the other is equivalent to encrypting and then decrypting. This restriction could be eliminated at the cost of a small increase in the size of the sealed object by changing the opener to be *List[Seal[Seal[dk.key, ka], Submaster[kb]]]*.

```
Key-And[ka: Key, kb: Key / dk: Key] ← Prog[ [];
        dk ← Create[Derived-Key];
        dk.key ← Create-Base-Key[];
        dk.opener ← List[Seal[Seal[dk.key, ka], kb]]];
        Return[dk];
        ];
```

Key-Or is implemented in the same manner as *Key-And*. One of the elements of the opener of *Key-Or* can be unsealed with a key set S only if S >> *ka* or S >> *kb*.

```
Key-Or[ka: Key, kb: Key / dk: Key] ← Prog[ [];
        dk ← Create[Derived-Key];
        dk.key ← Create-Base-Key[];
        dk.opener ← List[Seal[dk.key, ka], Seal[dk.key, kb]];
        Return[dk];
        ];
```

Key-Quorum creates a new base key and then splits the key into *n* pieces, where *n* is the length of the input key list. Things are arranged so that any combination of *q* of these pieces can be used to reconstruct the new base key. The *n* parts of the base key are then sealed with the *n* input keys, creating an opener that will yield the base key when a key set is available that unseals *q* or more distinct input keys.

```
Key-Quorum[q: Integer, kl: List[Key] / dk: Key] ← Prog[ [];
        dk ← Create[Derived-Key];
        dk.key ← Create-Base-Key[];
        -- Create the proper number of key pieces.
        dk.opener ← Threshold-Split[Encode[dk.key], Length[kl], q];
        -- Seal the pieces with the elements of kl.
        dk.opener.pieces ← Seal-List[dk.opener.pieces, kl];
        Return[dk];
        ];
```

Submaster is implemented by creating a new base key, and including the key in an opener sealed with *k*.

```
Submaster[k: Key / dk: Key] ← Prog[ [];
        dk ← Create[Derived-Key];
        dk.key ← Create-Base-Key[];
        dk.opener ← List[Seal[dk.key, k]];
        Return[dk];
        ];
```

Seal-Only ensures that only sets that unseal *k* will unseal *dk*. This derived key is set up to

encrypt objects with one half of a key pair, and to hide the other half in an opener. The opener can only be unsealed with a key set S if $S >> k$.

```
Seal-Only[k: Key / dk: Key] ← Prog[ [kp: Key-Pair];
        kp ← Create-Key-Pair[];
        dk ← Create[Derived-Key];
        dk.key ← kp.keya;
        dk.opener ← List[Seal[kp.keyb, k]];
        Return[dk];
        ];
```

Create-Indirect-Key creates a key that can be changed. This is accomplished by creating a derived key with an indirect opener. By changing the indirect opener, one can change the keys that unseal the indirect key.

```
Create-Indirect-Key[k: Key, tc: TC / dk: Key] ← Prog[ [];
        k ← Submaster[k];
        -- create indirect key
        dk ← Create[Derived-Key];
        dk.key ← k.key;
        dk.opener ← Create-Indirect[k.opener, tc];
        Return[dk];
        ];

Change-Indirect-Key[dk: Key, nk: Key, tc: TC] ← Prog[ [];
        -- if nk is NIL, delete key
        IF Null[nk] THEN
                Change-Indirect[dk.opener, NIL, tc];
        ELSE
        -- replace opener with new one
                Change-Indirect[dk.opener, List[Seal[dk.key, nk]], tc];
        Return[];
        ];
```

RUnseal and NA-Unseal are implemented as the following functions.

```
RUnseal[sx: Any, ring: List[Key], tc: TC / x: Any] ← Prog[ [];
        -- recursive unseal
        x ← Unseal[sx, ring, tc];
        IF Is[x, Sealed] THEN Return[RUnseal[x, ring, tc]];
        Return[x];
        ];

NA-Unseal[sx: Any, ring: List[Key], tc: TC / x: Any] ← Prog[ [];
        -- unauthenticated unseal
        -- sx does not have to be sealed
        x ← IF Is[sx, Sealed] THEN RUnseal[sx, ring, tc] ELSE sx;
        Return[x];
        ];
```

7.2.3 Strength

We reason about the strength of the protection mechanism in two parts. First, assuming that the cryptographic systems we use are perfect, we demonstrate the security and authentication properties of the mechanism. Second, we examine the assumption that cryptographic systems are perfect, and suggest usage conventions that would make the protection mechanism less susceptible to cryptanalysis.

7.2.3.1 Correctness Argument

It is difficult to prove facts about systems that are based on cryptography because, as we discussed, it is difficult to show that cryptosystems have certain properties. Thus, what we will do is to make strong assumptions about the behavior of the cryptographic systems that we use, and show that the secrecy and authentication properties of the protection mechanism follow from these assumptions.

First, we demonstrate the secrecy property of the protection mechanism, which is:

> **Secrecy.** *x* can be recovered from Seal[x, k] with a set of keys S if and only if S >> *k*.

Our first assumption has to do with the security of cryptography:

> **PC.** (Perfect Cryptography) Encrypt[x, k] reveals no information about *k*. If *k* is one half of a public-key pair, then *x* can only be recovered with the other half of the public-key pair, and if *k* is a classical key, then *x* can only be recovered with *k*.

We interpret "can only be recovered" to mean a total lack of information in the information theoretic sense. PC is close enough to what is expected of a practical cryptosystem to make it a reasonable assumption. However, we know of one exception to PC. To recover *x* from Encrypt[Encrypt[x, k1], k2] should always require a key set S, such that S >> *k1* and S >> *k2*. In the public-key system proposed by [Rivest et al. 78] if *k1* and *k2* are a public-key pair, then Encrypt[Encrypt[x, k1], k2] is *x*. In Key-And, where we will need to reason about an object that has been sealed twice, we will assume that *k1* and *k2* are not a public-key pair.

Our demonstration that the secrecy property follows from PC proceeds by induction on the structure of the key *k*. Our basis is the case where *k* is a base key.

> **Basis.** Assume that k is the result of Create-Base-Key or Create-Key-Pair. Seal[x, k] is transformed to Encrypt[Add-Checksum[x], k]. The secrecy property follows from PC.

We now assume the secrecy property as our induction hypothesis, and consider each way that derived keys can be created as our induction step.

> **Key-And.** Assume *dk* is the result of Key-And[ka, kb]. *dk* is
>
> [k, [Seal[Seal[k, ka], kb]]]
>
> where k is the result of a Create-Base-Key operation. Seal[x, dk] is
>
> [Seal[x, k], [Seal[Seal[k, ka], kb]]].
>
> By the induction hypothesis we need *k* to recover *x*, and to recover *k* we need a set of keys that is admitted to *ka* and to *kb*. Thus, a set of keys S can recover *x* if and only if

((S >> kb) ∧ (S >> ka)) ∨ (dk ∈ S).

This is precisely the unseals relation for Key-And.

Key-Or. Assume *dk* is the result of Key-Or[ka, kb]. *dk* is

[k, [Seal[k, ka], Seal[k, kb]]]

where k is the result of a Create-Base-Key operation. Seal[x, dk] is

[Seal[x, k], [Seal[k, ka], Seal[k, kb]]].

By the induction hypothesis we need *k* to recover *x*, and to recover *k* we need a set of keys that is admitted to *ka* or to *kb*. Thus, a set of keys S can recover *x* if and only if

(S >> ka) ∨ (S >> kb) ∨ (dk ∈ S).

This is precisely the unseals relation for Key-Or.

The arguments for Submaster and Create-Indirect are very similar to Key-And and Key-Or, and thus we will omit them.

Seal-Only. Assume *dk* is the result of Seal-Only[k]. *dk* is

[ka, [Seal[kb, k]]]

where [ka, kb] is the result of Create-Key-Pair. Seal[x, dk] is

[Seal[x, ka], [Seal[kb, k]]].

By the induction hypothesis we need *kb* to recover *x*, and to recover *kb* we need a set of keys that is admitted to *k* . *kb* only appears in openers. Thus, a set of keys S can recover *x* if and only if

(S >> k).

This is precisely the unseals relation for Seal-Only.

Key-Quorum. Assume *dk* is the result of Key-Quorum[q, List[k_1 .. k_n]]. *dk* is

[k, [Seal[p_1, k_1] ... Seal[p_n, k_n]]]

where *k* is the result of a Create-Base-Key operation, and p_1 ... p_n are the result of Threshold-Split[k, n, q]. Seal[x, k] is

[Seal[x, k], [Seal[p_1, k_1] ... Seal[p_n, k_n]]].

By the induction hypothesis we need *k* to recover x. To recover *k* we need *q* distinct values of p_1 ... p_n. By the induction hypothesis we can only recover *q* distinct values of p_1 ... p_n with a set of keys that is admitted to *q* distinct keys in k_1 ... k_n. Thus, a set of keys S can recover *x* if and only if for some set of *q* distinct keys k_a ... k_q drawn from k_1 ... k_n:

((S >> k_a) ∧ ... ∧ (S >> k_q)) ∨ (dk ∈ S).

This is precisely the unseals relation for Key-Quorum.

□.

We will now demonstrate the authentication property of the protection mechanism, which is:

> **Authentication.** If *x* was not sealed with a key that is unsealed by *S* then the result of Unseal[x, S] will be Error['Failed].

To demonstrate the authentication property we need to make an assumption concerning the behavior of checksums:

> **CHK.** Assuming *x* and Encrypt[x, k] are known, it is intractable to compute Encrypt[Add-Checksum[x], k] unless *k* is known.

The argument we use is once again based on induction on the structure of a key that was used to seal an object. As our basis we will assume that Unseal decides that an object has been sealed with a base key.

> **Basis.** Assume that Unseal decides that *x* has been sealed with base key *k*. By CHK, unless the client that sealed *x* had *k* it could not generate the correct encrypted checksum. If the checksum of *Decrypt[x, kl]* is not valid Unseal returns *Error['Failed]*.

Now we assume the authentication property as the induction hypothesis, and show that if an object was sealed with a derived key the authentication property is also guaranteed.

> **Induction Step.** Assume that Unseal decides that *x* has been sealed with a derived key. It unseals what it thinks is an opener with *kl*. If the opener was sealed a key that is unsealed by *kl*, then this operation will return a new key list *kl'*. By the induction hypothesis this operation would return *Error['Failed]* if *kl'* had not been sealed by a key that is unsealed by *kl*. Unseal then uses *kl* and *kl'* to unseal *x*. By the induction hypothesis if *x* was sealed by a key that is not unsealed by the union of *kl* and *kl'* then Unseal will return *Error['Failed]*. A key that is unsealed by *kl* was required to create the opener that contained *kl'*. Thus, if *x* was not sealed with a key that is unsealed by *kl* then the result of Unseal[x, kl] will be *Error['Failed]*.

□.

7.2.3.2 Susceptibility to Cryptanalysis

In the previous section we assumed that cryptography is perfect. Of course it is not. Often, breaking a practical cryptographic system is a matter of economics [Diffie and Hellman 77]. However, if some guidelines are followed, the susceptibility of our protection system to cryptanalysis can be reduced.

The cryptosystems that are used must be secure against a known-cleartext attack. The checksums that are included in sealed objects and the implementation of Key-And increase the probability that an intruder will be able to use a known-cleartext attack.

It is wise to keep the amount of information protected with a single key to a minimum. This makes it more difficult for an intruder to perform a known-cleartext attack, and it reduces the vulnerability of the system to a single cryptanalytic success. Because keys are inexpensive, a client should use as many keys as are natural for its application.

The strength of the cryptographic system used to protect information should correspond to the length of time the information needs to be kept secret. For example, if certain information is going

to be sensitive for twenty years, it would not be wise to protect it with a cryptosystem that can be broken in one year. Another metric is that the value of information protected with a cryptographic system should be considerably less than the cost of an attack on the system. Cryptographic sealing can accommodate the coexistence of a number of cryptosystems that have different key sizes and strengths. Thus, the strength of a cryptosystem can be matched to the sensitivity of the information it protects.

Finally, there is no need to divulge information that might lead to a successful cryptanalytic attack to clients that do not need to know the information. For example, public keys can be protected from clients that do not need them.

7.3 Applications

We now turn our attention to applications of cryptographic sealing. The first two sections show how cryptographic sealing and a simple active protection mechanism can implement a variety of popular protection mechanisms, including capabilities, access control lists, and information flow control. The third section demonstrates secure processors. The last section shows how secure processors can be used to implement revocation.

7.3.1 Privilege Establishment

7.3.1.1 Key Rings

Keys are used to represent privileges, and thus a list of keys defines a set of privileges. Each user has a personal list of keys, or *key ring*, that defines his privileges. When a key ring is stored, it is sealed with a key that only its owner knows. A user authenticates himself to the system by providing his key ring key. The key ring key is used to unseal the user's key ring in his processor, resulting in his list of privileges.

A key ring key could be stored on a magnetic card, or perhaps transformed into an easily remembered sentence, such as "Ralph and George ran to the store on a rainy cold day with their Aunt Essie's dog Fred". Such a transformation could be accomplished with a context free grammar.

A user's unsealed key ring is the third argument to Open, *ring*. Open is very careful with a user's key ring, and will not transmit it to any other processor.

7.3.1.2 Encrypted Objects

Cryptography is used to control access to information stored in files and indexes. An *encrypted object* E is an object that is encrypted with a certain conventional key K. Thus, possession of K gives a client the ability to access the information in E. Objects may be encrypted many times.

Create-Encrypted[ref: Reference, k: Key / eref: Encrypted]

> Create-Encrypted creates an encrypted object. If the object is a file or an index, then to access the object a client must have a key that is admitted to *k*. Create-Encrypted assumes that the referent of *ref* will be totally overwritten and currently contains no information.

Encrypted objects are implemented by creating a special reference that contains their key and a reference to their encrypted form.

```
Encrypted ← Record[ref: Reference, key: Key];

Create-Encrypted[ref: Reference, key: Key / eref: Encrypted] ← Prog[ [];
        eref ← Create[Encrypted];
        eref.ref ← ref;   eref.key ← key;
        Return[Add-Type[eref, Major-Type[ref]]];
        ];
```

When an encrypted object is opened, a class structure is set up that will encrypt and decrypt the data portions of read and write requests.

```
Open-Encrypted[ref: Reference, tc: TC, ring: List[Key], guards: List[Key] / c: Class]
  ← Prog[ [k: Key; ew: Function];
      -- try to unseal key
      k ← NA-Unseal[ref.key, ring, tc];
      -- failed, client does not have access
      IF Is[k, Error-Type] THEN Return[k];
      ew ← IF Is[ref, Index] THEN 'Encrypted-Index-Write ELSE 'Encrypted-File-Write;
      c ← Open[ref.ref, tc, ring, guards];
      IF Is[c, Error-Type] THEN Return[c];
      c ← Create-Class[List[
            'CopyReference, 'Default-Copy,
            'Read, 'Encrypted-Read,
            'Write, ew,
            'Enumerate, 'Encrypted-Enumerate], c];
      -- Instance variables: k, ref
      Return[c];
      ];

Encrypted-Read[/ value: Byte-Array] ← Prog[ [];
      value ← Apply[superclass, request];
      Return[Conventional-Decrypt[value, k.key]];
      ];

Encrypted-Index-Write[entry-name: Byte-Array, value: Byte-Array] ← Prog[ [];
      IF Null[value] THEN Return[superclass | Write[key, NIL]]
      ELSE Return[superclass | Write[entry-name, Conventional-Encrypt[value, k.key]]];
      ];

Encrypted-File-Write[startpage: Integer, pages: Integer, value: Byte-Array] ← Prog[ [];
      Return[superclass | Write[startpage, pages, Conventional-Encrypt[value, k.key]]];
      ];

Encrypted-Enumerate[last: Entry / next: Entry] ← Prog[ [];
      IF Not[Null[last]] THEN last.value ← Conventional-Encrypt[last.value, k.key];
      next ← superclass | Enumerate[last];
      next.value ← Conventional-Decrypt[next.value, k];
      Return[next];
      ];
```

7.3.1.3 Guarded Objects

To provide integrity and availability we introduce a simple active protection mechanism. Imagine that each object is assigned a unique set of passwords, one for each of its independent privileges. We will call these passwords *guards*. Because each object has a unique set of guards, they must be stored with the object. For example, suppose file *F* is assigned write guard *G*. The processor that stores *F* would require that *G* be presented for each write access.

Guards are presented in the fourth argument to Open, *guards*. Guards are checked at open time. For example, if a reference is opened for update (it is not a Read-Only reference) then Open checks for a write guard if one is required. Guards can be directly manipulated by a client, or the facilities described below can be used to help manage guards.

The system defines a standard set of guard types. An Access guard must be provided before an

object can be used in any way, a Read guard must be provided before an object can be read, a Write guard must be provided before an object can be written, a Create guard must be provided before an object will service a create operation, and a Change guard must be provided before an object will allow its guards to be changed. An object can implement a subset of these guard types, or it can choose to implement guards based on its special needs.

c: Class | Set-Guard[gt: Type, gk: Simple-Key]

Set-Guard sets a guard for the object serviced by *c* to be *gk*. The guard will protect the set of privileges specified by *gt*. If *gk* is NIL Set-Guard removes guards.

ic: Index-Class | Set-Entry-Guard[entry-id: Byte-Array, gt: Type, gk: Simple-Key]

Index entries can also be protected by guards. Set-Entry-Guard sets a guard for the index entry specified by *entry-id*. The guard will protect the set of privileges specified by *gt*. If *gk* is NIL Set-Entry-Guard removes guards.

7.3.1.4 Protected Volumes

In many cases clients would like to have the files that they create protected automatically. To this end we provide the notion of a *protected volume*. Files created on a protected volume assume a default protection structure specified by the protected volume.

Create-Protected-Volume[ref: Reference, tl: List[Type], kl: List[Key] / pref: Protected]

Create-Protected-Volume creates a reference for a protected volume. All files created on the protected volume will have the types of access controls in *tl* set, and the resulting privilege keys will be sealed with corresponding elements of *kl*. The elements of *tl* may be guard types, or they may be the type *Encrypted*, in which case the file will be encrypted.

```
Protected ← Record[ref: Reference, tl: List[Key], kl: List[Key]];

Create-Protected-Volume[ref: Reference, tl: List[Key], kl: List[Key] / pref:
Protected]
    ← Prog[ [];
        pref ← Create[Protected];
        pref.tl ← tl;   pref.kl ← kl;   pref.ref ← ref;
        Return[Add-Type[pref, Major-Type[ref]]];
        ];
```

When a protected volume is opened the model implementation establishes a class structure that will set protection controls in response to create file operations, and that will properly copy a protected volume reference.

```
Open-Protected[ref: Reference, tc: TC, ring: List[Key], guards: List[Key] / c: Class]
  ← Prog[ [];
      c ← Open[ref.ref, tc, ring, guards];
      IF Is[c, Error-Type] THEN Return[c];
      Return[Create-Class[List[
              'CopyReference, 'Default-Copy,
```

```
            'Create-File, 'Protected-Create], c]];
        ];
Protected-Create[/ cref: Capability] ← Prog[ [];
        [k: Key; cl: Class;  priv: List[Sealed]; x: Cons[Type, Key]];
        priv ← NIL;
        -- create file
        cref ← Apply[superclass, request];
        -- open file
        cl ← Open[cref, tc, ring];
        -- apply guards or encrypt file
        FOR x IN Pair[ref.tl, ref.gl] DO [
                k ← Create-Base-Key[];
                IF Is[car[x], Encyrpted] THEN
                        cref ← Create-Encrypted[cref, Seal[k, cdr[x]]];
                ELSE [
                        cl | Set-Guard[car[x], k];
                        priv ← Append[priv, Seal[k, cdr[x]]];
                        ];
                ];
        cl | Close[];
        -- Capabilities are described in Section 7.3.2.1
        Return[Create-Capability[cref, priv]];
        ];
```

7.3.2 Common Protection Mechanisms

All of the ingredients are now at hand for creating a large number of common protection mechanisms. We have represented privilege by the possession of keys, and these keys can be sealed such that only authorized clients can unseal them. We will treat the three major types of protection mechanisms in current use: capabilities, access control lists, and information flow control.

7.3.2.1 Capabilities

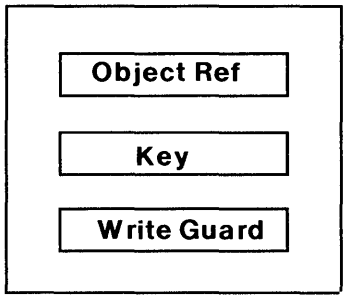
A capability is an unforgable ticket which permits a possessor to access the object it names with certain privileges [Dennis and Van Horn 66, Lampson 69]. No special privileges are required in our system to make a copy of a capability.

Capabilities can be implemented by including a set of keys and guards in an object's reference. For example, imagine object *O* is encrypted with the conventional key *K*, and *G* is the write guard for object *O*. If a reference for *O* includes *K*, then it can be used to read *O*. If the reference also includes *G*, it can be used to read or write *O*. Without *G* or *K* it is impossible to read or write *O*. To the extent that keys and guards are considered impossible to guess, capabilities can be considered unforgable. A capability reference is shown in Figure 7.5.

Create-Capability[ref: Reference, pl: List[Key] / cref: Capability]

> Create-Capability creates a capability for ref. *pl* is the list of keys that defines the set of privileges that the new capability will have. If any elements of *pl* are sealed, an attempt will be made to unseal them when the capability is opened.

```
Capability ← Record[ref: Reference, pl: List[Key]];
```



Capability Reference

Figure 7.5

```
Create-Capability[ref: Reference, pl: List[Key] / cref: Capability] ← Prog[ [];
        cref ← Create[Capability];
        cref.pl ← pl;
        cref.ref ← ref;
        Return[Add-Type[cref, Major-Type[ref]]];
        ];
Open-Capability[ref: Reference, tc: TC, ring: List[Key], guards: List[Key] / c: Class]
    ← Prog[ [pl];
        pl ← Normalize[ref.pl, tc];
        pl ← NA-Unseal-List[pl, Append[ring, guards], tc];
        -- expand privilege keys by pl
        c ← Open[ref.ref, tc, Append[ring, pl], Append[guards, pl]];
        IF Is[c, Error-Type] THEN Return[c];
        Return[Create-Class[List['CopyReference, 'Default-Copy], c]];
        ];
```

7.3.2.2 Access Control Lists

An access control list system associates a list of users with each object. This list describes who may access the object, and with what privileges [Saltzer and Schroeder 79].

To implement access control lists each user of the system creates a key pair, and makes one of the keys of this pair public. Section 7.4.2 discusses how one user can reliably learn another user's public key. A user keeps the private half of his key pair on his key ring. Thus if user *X* seals an object with user *Y's* public key, only user *Y* will be able to unseal it. This result follows directly from the public-key cryptography that is used to implement key pairs.

Access control lists are implemented by sealing the privileges in a capability reference, as shown in Figure 7.6. In general, a key or guard is sealed with the *Key-Or* of the users' keys that have been granted the corresponding privilege. Once sealed, these references can be placed in a public directory system. Only users that have been granted a privilege will be able to unseal its corresponding key or guard.

It is often desirable to be able to grant privileges to a group of users and allow the members of the group to change over time. If revocation is not required, indirect keys can be used to define a group as

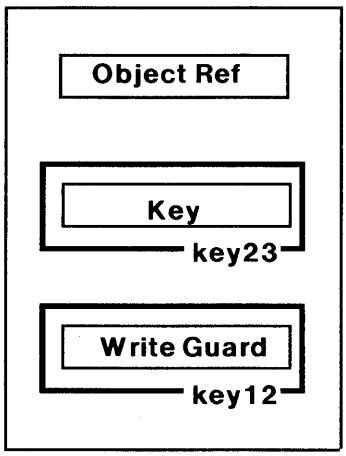
```
Group-Key ← Seal-Only[Create-Indirect-Key[
        Key-Or[u1 ... Key-Or[un-1, un]]]];
```

where u_1 ... u_n are the users' keys that are members of the group. The members of the group can be altered with *Change-Indirect-Key.*

7.3.2.3 Information Flow Control

In an information flow control scheme [Denning 76] each object is labeled with one or more classifications, and the output of a computation is labeled with the union of the classifications of its inputs. An example of an information flow control scheme is the military system of classification. If *Top Secret* and *Crypto* information are used in a report, then the report would be classified *Top Secret, Crypto.*

Information flow control schemes are usually nondiscretionary, meaning that the classification of



Read
<Jones>, <Smith>, <Zephyr>

Write
<Jones>, <Smith>

Key23 = Key-Or[<Jones>, <Smith>, <Zephyr>]

Key12 = Key-Or[<Jones>, <Smith>]

Object Reference: Access Control List

Figure 7.6

a computation's output is fixed, and can not be altered by a client. In the information flow control scheme we present a client chooses the classifications of its outputs. Once a classification is specified, the protection mechanism ensures that it is properly enforced.

Cryptographic sealing can be used to implement information flow control in the following manner. Represent each classification by a key. If O is created from objects that have classifications C_1 ... C_n, seal the privileges in O's reference with *Key-And[*C_1 ... *Key-And[*C_{n-1}, C_n*]]*.

For example, imagine that the fictitious company Sierra has two divisions, a medical division and an office division. Sierra would like to enforce the policy that information that is private to one division is only accessible to employees of that division. Furthermore, Sierra has financial information that only senior managers are allowed to access. Sierra could create keys to represent these classifications as follows:

```
Medical ← Seal-Only[Create-Indirect-Key[
        Key-Or[<adams>, ... Key-Or[<thatch>, <west>]]]];

Office ← Seal-Only[Create-Indirect-Key[
        Key-Or[<jones>, ... Key-Or[<rainbow>, <smith>]]]];

Financial ← Seal-Only[Create-Indirect-Key[
        Key-Or[<adams>, ... Key-Or[<irby>, <welch>]]]];
```

The names in brackets represent the public keys of people that are members of each classification. Indirect keys are used so that membership of each classification group can be altered later. Information about the overall performance of Sierra could be sealed with *Financial.* Information about the financial performance of the medical division could be sealed with *Key-And[Financial, Medical].*

It is of course possible to combine access control lists and information flow control in a single system. This hybrid structure results when "the need to know" is added to an information flow control policy.

7.3.3 Secure Processors

By a *secure processor* we mean two things. First, we must have confidence that a secure processor will not inadvertently disclose secret information. Specifically, the very general mechanism for remote evaluation we introduced in Chapter 2 must be limited in some way to keep intruders from executing arbitrary functions on the processor of their choice. Second, when a conversation is started with a secure processor the identity of the secure processor must be authenticated, and the conversation must be kept secret. For example, when a processor sends a capability to another processor, the exchange must be kept secret, and the sending processor must be sure that it is transmitting the capability to its intended recipient, not to an intruder.

7.3.3.1 Limiting Remote Evaluation

If a processor uses cleartext forms of encrypted objects it must be demonstrated that its remote evaluation mechanism will not inadvertently disclose information. The simplest solution to this problem is to forbid any processor that processes cleartext from entertaining remote requests. In such a scenario only shared processors would accept remote requests, and these shared processors would only deal with encrypted objects. Even if a read guard was not enforced properly, an

intruder would not be able to make sense of the information he received. Unfortunately this simplistic approach can not always be employed.

We limit the flexibility of the remote evaluation mechanism by controlling the environment in which remote evaluations are performed. The Eval in Request-Eval is done with respect to an environment that only contains the functions Open-Connection, Close-Connection, and Remember-Eval. The Eval in Remember-Eval is done with respect to an environment that only contains the functions Open-Door and Door-Eval. Finally, the Eval in Door-Eval is done with respect to an environment that only contains the function Open. If global functions are used to register transaction participants (Section 3.1.2) these functions will have to be added to the environment of Request-Eval.

This limits remote clients to only being able to access objects for which they have references. Although we have not done so in the model implementation, Open could carefully ignore malformed references. Although invented references would not give an intruder access to private objects, the intruder may be able to cause Open to do strange things.

7.3.3.2 Secure Channels

We describe here a method for creating *secure, one-way authenticated* communication channels. Secure means that an intruder can not discover the information that is being transmitted, and can not "spoof", or pretend that he is one of the participants in the conversation. One-way authenticated means that the client knows that it will be connected to the secure processor it specifies. Secure one-way authenticated channels are provided by secure processors.

A reference for a secure processor includes an ordinary reference for the processor, and a public key that will be used to initiate conversations with the processor. The public key is half of a key pair, and is not admitted to itself. We assume a secure processor can obtain the other half of this pair by evaluating Get-Processor-Key.

```
Secure-Processor ← Record[proc: Processor, public-key: Key];
```

A secure processor is implemented with the help of a *secure door.* A secure door is an object implemented by a secure processor that services Eval requests. Secrecy is maintained by encrypting all messages to and from a secure door. The key that is used for this encryption is contained in the reference for the secure door. Intruders can not discover this key, because it is sent to the secure processor sealed with the processor's public key. Authentication of the secure processor is provided by virtue of the fact that only it will be able to unseal the key and carry on a conversation.

```
Secure-Door ← Record[k: Sealed[Key]];

Open-Secure-Processor[ref: Reference, tc: TC, ring: List[Key], guards: List[Key] / c: Class]
    ← Prog[
        [door-class: Class; seqa: Integer; seqb: Integer; sl: Lock; k: Key; sd: Secure-Door;
        unique-id: UniqueID];
        -- two sequence numbers, one for transmit, one for receive
        seqa ← 0;   seqb ← 0;
        sl ← Create-Lock[];
        k ← Create-Base-Key[];
        sd ← Create[Secure-Door];
        sd.k ← Seal[k, ref.public-key];
```

```
door-class ← Open[Create-Located[sd, ref.proc], tc, ring, guards];
IF Is[door-class, Error-Type] THEN Return[door-class];
-- ask secure door for the connection identifier
unique-id ← door-class | Connection-ID[];
c ← Create-Class[List[
        'Eval, 'Secure-Processor-Eval,
        'CopyReference, 'Default-Copy], NIL];
Return[c];
];
```

Spoofing is prevented with two mechanisms. First, every time a secure door is opened its secure processor provides a unique identifier that is incorporated into every message. If an intruder replayed a secure conversation after the conversation had ended, a new unique identifier would be assigned to the conversation, and the recorded conversation would be ignored. Second, every message to or from a secure door includes a message sequence number. This number is checked, and ensures that an intruder does not replay portions of a conversation while it is still in progress.

```
Secure-Message ← Record[seq: Integer, id: UniqueID, message: Any];

Create-Message[message: Any / sm: Sealed[Secure-Message]] ← Critical[sl, 'Prog[ []];
        sm ← Create[Secure-Message];
        sm.seq ← seqa;
        sm.id ← unique-id;
        sm.message ← message;
        seqa ← seqa + 1;
        Return[Seal[sm, k]];
        ]];

Unseal-Message[sm: Sealed[Secure-Message] / message: Any] ← Critical[sl, 'Prog[ []];
        sm ← Unseal[sm, List[k]];
        IF Is[sm, Error-Type] THEN Return[sm];
        IF And[sm.seq=seqb, sm.id=unique-id] THEN [
                seqb ← seqb + 1;
                Return[sm.message];
                ];
        Return[Error['Tampering]];
        ]];

Secure-Processor-Eval[form: Any / result: Any] ← Prog[ []];
        result ← door-class | Eval[Create-Message[form]];
        Return[Unseal-Message[result]];
        ];
```

Imagine object O is located with a secure processor SP. Here is what happens when O is opened:

1. Open-Located (Section 3.2.4) causes SP to opened. When SP is opened, Open-Secure-Processor opens a secure door SD at the remote processor. Open-Secure-Door creates an environment that includes an empty set called Doors.

2. Open-Located then opens O with Open-Door through SP The class for SP forwards the Open-Door request to the remote processor, to the class for SD. The Open-Door is thus done in the context of SD. Open-Door creates a door to O (Section 3.2.4) in SD's set Doors.

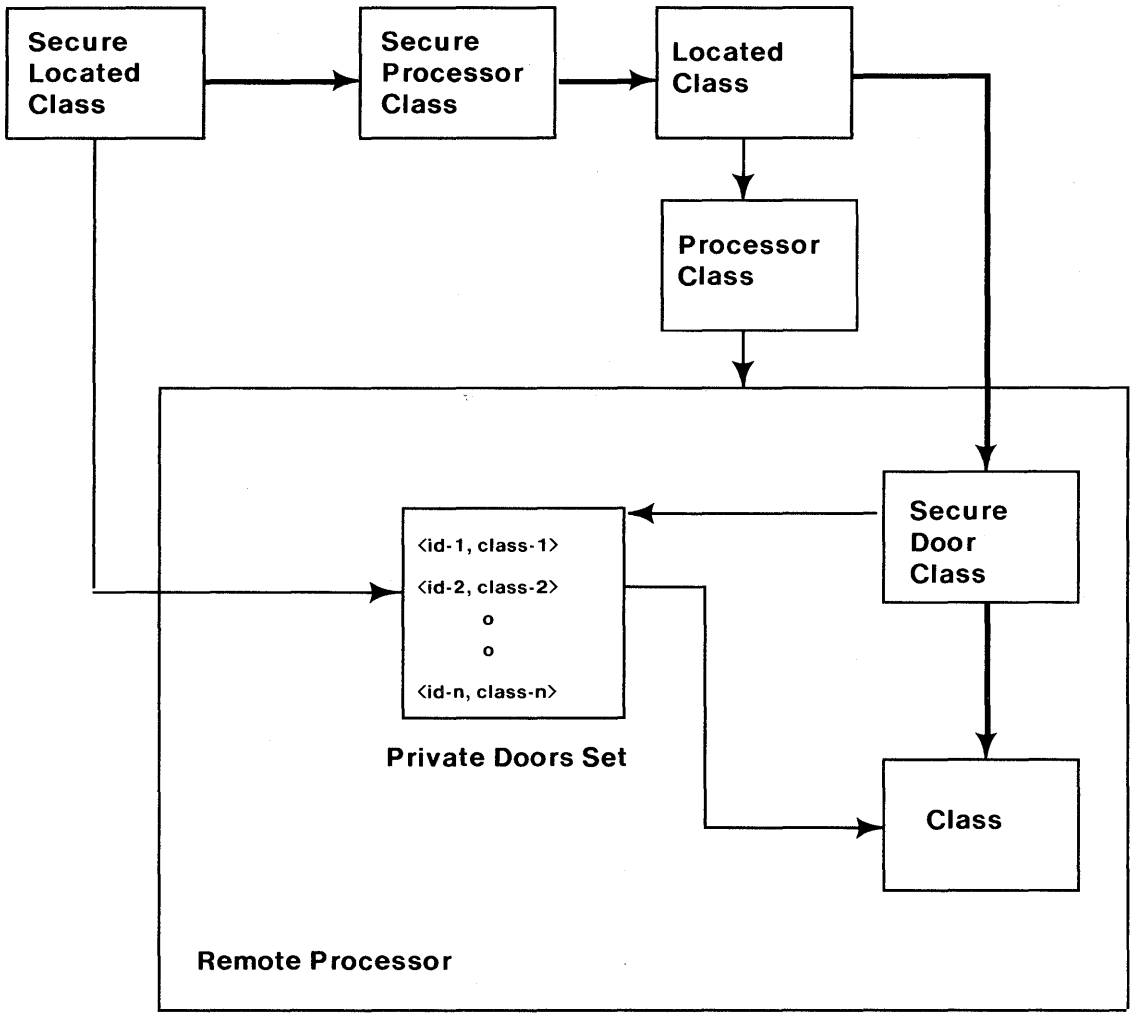
Thus, it is impossible for an intruder to access the door to O without first passing through the secure door. This is to prevent an intruder from probing the space of door identifiers and bypassing our security.

Figure 7.7 shows the class structure that is established when O is opened. The heavy line in the figure shows the flow of control when a request is made to the referent of O.

```
Open-Secure-Door[ref: Reference / class: Class] ← Prog[
        [seqa: Integer; seqb: Integer; unique-id: Unique-ID;  Doors: Set; k: Key; sl:
        Lock];
        -- set sequence counters to zero and initialize connection identifier
        seqa ← 0;   seqb ← 0;
        unique-id ← GetUniqueID[];
        sl ← Create-Lock[];
        -- create private Doors set
        Doors ← Create-Set[];
        -- Unseal key for this channel
        k ← Unseal[ref.key, Get-Processor-Key[]];
        Return[Create-Class[List['Eval, 'Secure-Door-Eval, 'Connection-ID, 'Connection-ID],
        NIL]];
        ];
Connection-ID[/ id: UniqueID] ← Prog[ [];
        -- return identifier of secure connection
        Return[unique-id];
        ];

Secure-Door-Eval[in: Sealed[Secure-Message] / out: Sealed[Secure-Message]] ← Prog[
        [result: Any];
        in ← Unseal-Message[in];
        IF Is[in, Error-Type] THEN Return[Error['Discard]];
        -- The environment of Eval should only include
        --     Open-Door and Door-Eval
        result ← Eval[in.message];
        Return[Create-Message[result]]];
        ];
```

7.3.4 Revocation

The problem of revoking access to an object once it has been granted is known as revocation [Redell 74]. An indirect key does not provide revocation. Clients can hold on to the contents of indirect openers, and thus continue to access items sealed with the indirect key regardless of how the opener might be updated. Revocation is implemented by a trusted secure processor that provides a level of indirection between a client and the unsealed form of an object's privilege keys.



Structure of a Secure Located Class

Figure 7.7

Create-Revocable-Capability creates a capability that has certain privileges, and these privileges can be changed by Change-Revocable-Capability. If a client's privileges are reduced by Change-Revocable-Capability, on subsequent calls to Open there is no way the client can assume its old privileges.

Create-Revocable-Capability[ref: Reference, pl: List[Key], key-list: List[Key], ck: Key, i: Index, sp: Secure-Processor, tc: TC / rref: Revocable]

> Create-Revocable-Capability creates a new revocable capability. *pl* is the set of privileges that the capability has, and these privileges are protected by *key-list* with Seal-List. Clients that can unseal *ck* can change the revocable capability. The index and secure processor specified are used to implement the capability, and must be trusted.

Change-Revocable-Capability[rref: Revocable, pl: List[Key], key-list: List[Key], ring: List[Key], tc: TC]

> Change-Revocable-Capability changes *rref* such that the new set of privileges is *pl*, and these permissions are protected by *key-list* with Seal-List. A key ring that can unseal *ck* must be supplied.

7.3.4.1 Protected Indirection

In order to implement revocation we first need to introduce indirection with access protection. The functions shown below are a simple variation of the standard indirection primitives. As an exercise the reader might consider how to implement these functions from the primitives we have introduced.

Create-Secure-Indirect[record: Any, index: Index, tc: TC, access: Key / ind: Indirect];

> Create-Secure-Indirect creates an indirect entry that can only be accessed by a client that can unseal *access*.

Change-Secure-Indirect[ie: Indirect, contents: Any, tc: TC, ring: List[Key]]

> Change-Secure-Indirect changes an indirect entry if *ring* unseals *access*, and returns NIL. Otherwise Error['Failed] is returned.

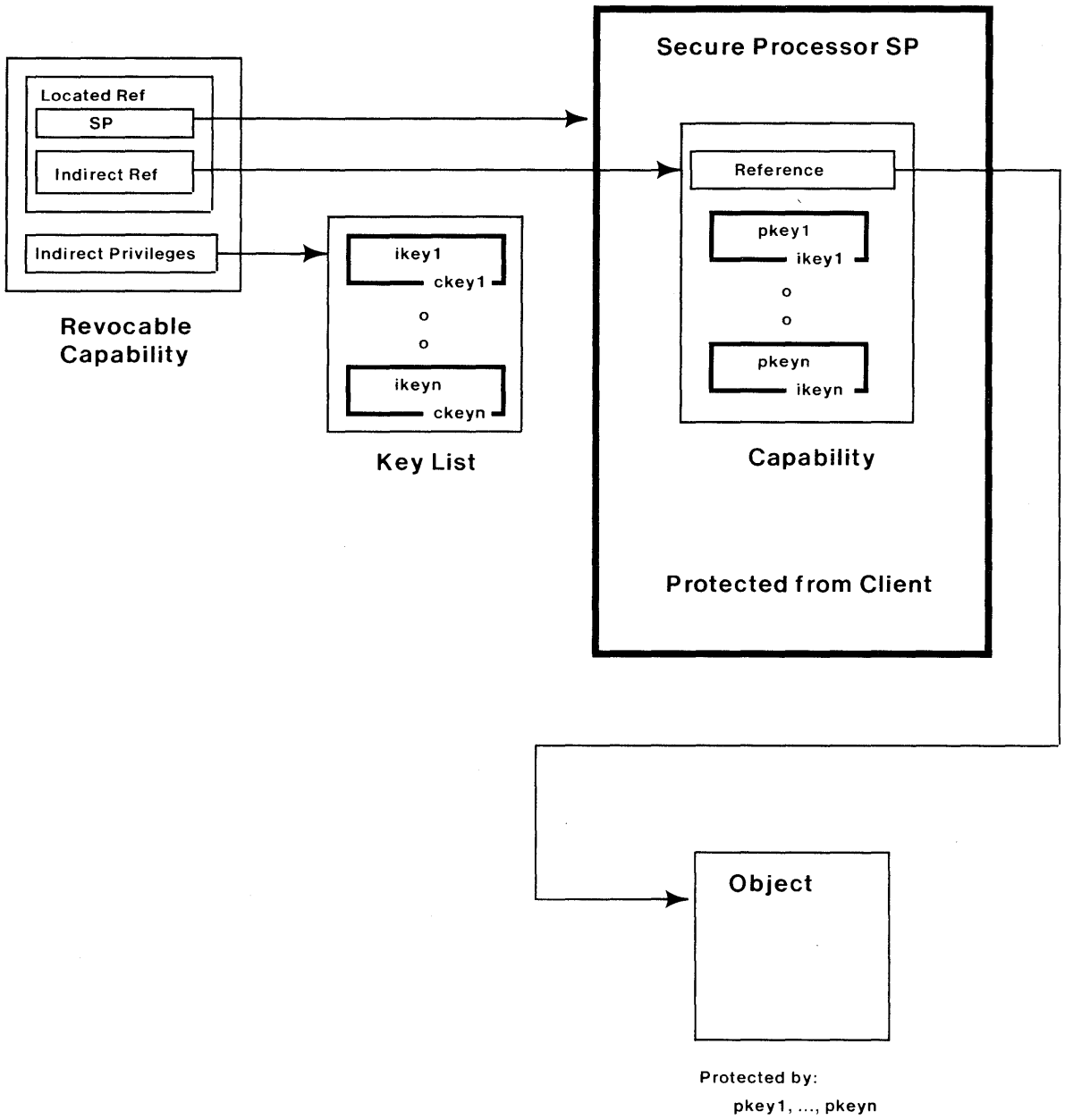
Lookup-Secure[ie: Indirect, tc: TC, ring: List[Key]]

> Lookup-Secure returns the contents of *ie* if *ring* contains a key that unseals *access*. Error['Failed] is returned if *ring* does not unseal *access*.

7.3.4.2 Revocation Algorithm

Figure 7.8 shows how revocation is accomplished. Assume that the client can not access the second capability shown, but secure processor SP can. The client's key ring is used to used to unseal the first capability, which yields a set of intermediate keys. These intermediate keys are then passed to SP. SP takes the set of intermediate keys and determines the privilege keys for the object. SP opens the object on the client's behalf with these keys. Intermediate keys are employed so a user's key ring does not have to be passed to SP.

The model implementation of revocation follows.



A Revocable Capability

Figure 7.8

```
Create-Revocable-Capability[ref: Reference, pl: List[Key], key-list: List[Key], ck: Key, i:
    Index, sp: Secure-Processor, tc: TC / rref: Revocable] ← Prog[
        [nkl: List[Key]; gate: List[Sealed[Key]]];
        nkl ← Create-Key-List[Length[pl]];
        gate ← Seal-List[pl, nkl];
        key-list ← Seal-List[nkl, key-list];
        -- with nkl gate can be unsealed
        rref ← Create-Capability[ref, gate];
        -- we assume that the secure processor will use its
        -- private key when it looks up indirect references.
        rref ← Create-Secure-Indirect[rref, i, tc, Key-Or[ck, sp.public-key]];
        -- switch to secure processor
        rref ← Create-Located[rref, sp];
        -- with key-list nkl can be unsealed
        rref ← Create-Capability[rref, Create-Indirect[key-list, i, tc]];
        Return[rref];
        ];

Create-Key-List[n: Integer / kl: List[Key]] ← Prog[ [i: Integer];
        -- creates a list of n keys
        kl ← NIL;
        FOR i FROM 1 TO n DO kl ← Cons[Create-Base-Key[], kl];
        Return[kl];
        ];

Change-Revocable-Capability[rref: Revocable, pl: List[Key], key-list: List[Key], ring:
    List[Key], tc: TC]] ← Prog[
        [nref: Reference;  oref: Reference; nkl: List[Key]; gate: List[Sealed[Key]]];
        -- create new set of intermediate keys
        nkl ← Create-Key-List[Length[pl]];
        -- create new sealed privilege list
        gate ← Seal-List[pl, nkl];
        -- with key-list, a client can discover intermediate keys
        key-list ← Seal-List[nkl, key-list];
        oref ← Lookup-Secure[rref.ref.ref, tc, ring];
        nref ← Create-Capability[oref.ref, gate];
        Change-Secure-Indirect[rref.ref.ref, nref, tc, ring];
        Change-Indirect[rref.pl, key-list, tc];
        ];
```

7.4 Practical Considerations

7.4.1 Changing Protection Controls

Once a set of protection controls has been established there are two ways of changing the controls. The first way is to create the protection structure with indirect keys. For example, if an indirect key is made for the users that can access a file, it is a simple matter to change this indirect key to authorize additional users. Another way to change protection controls is to reconfigure an object to a new implementor that is protected with a desired structure. The new implementing object may in fact be the same object as the old one, but protected in a new way.

7.4.2 Authentication in the Large

A practical system must ensure that external names are mapped into correct internal keys. There are a number of solutions to this problem. Needham and Schroeder [Needham and Schroeder 78] have demonstrated one way of transforming external names into authentic internal keys.

Here is a straightforward way to solve the problem in our framework. Provide every processor with a reference R that is located with a secure processor. Let R be a reference for the file that contains the root of the naming system. If a client uses R as the starting point in resolving names to references or keys, then the client will obtain an authentic internal reference or key. We assume that R is protected against malicious modification.

The problem with this approach is that a corrupt system administrator could change R's referent in such a way that users would unwittingly give him access to their objects. To guarantee that an intermediate party does not tamper with key distribution it is necessary to distribute personal public keys outside of the system.

Another approach would be to create a key pair, [keya, keyb]. *keya* is kept secret by the system administration, and <name, public-key> pairs in the system directory are sealed with *keya. keyb* is made public, and given to users (perhaps in a face to face meeting). It is possible for a user to tell if an entry in the public directory is authentic by unsealing it with *keyb.*

7.4.3 Performance

The performance of cryptographic sealing completely depends on the performance of its foundation. High performance VLSI components have lowered the cost of conventional cryptography to a point where it can be ignored for practical purposes. For example, an encryption unit built by the author for the Xerox Dolphin processor can encrypt or decrypt 512 bytes in 332 microseconds. However, the performance of public-key cryptography and threshold schemes may be a significant consideration. The time for a single public-key encryption or decryption is currently measured in fractions of a second. The following refinements are intended to improve the performance of cryptographic sealing when keys from Create-Key-Pair or Key-Quorum are used:

1. Unseal discards keys derived from openers as soon as they are no longer needed. An important optimization would be to remember these keys, and use them to reduce the number of future unseal operations. When such keys were not in use, they could be stored on a user's key ring.

2. When client X is about to seal an object with K, a key-pair half, it creates K',

   K' ← Submaster[K],

   and seals the object with K' instead of K. K' is then stored in a cache in X's secure processor. In the future if X is going to seal another object with K, it uses K' instead.

   Now imagine that client Y is unsealing the objects that client X has sealed in this manner. If client Y has adopted the previous suggestion, it will only have to perform a single public-key decryption to recover an entire set of sealed objects produced by X. This case commonly arises when one user is granting privileges to another user.

7.4.4 Comments

The protection primitives we defined could keep comments, so it would be possible to determine how something was sealed, even if one could not unseal it. This would be useful for determining how objects were protected.

7.4.5 Elimination of Authentication

In some cases the authentication property of the protection mechanism may not be required. In such instances the checksum in sealed objects could be eliminated, reducing their size.

7.5 Comparative Analysis

To understand how cryptographic sealing might be used it is important to understand its advantages and disadvantages when compared with traditional protection mechanisms. The disadvantages of cryptographic sealing are:

1. If integrity or availability guarantees are required, a supplemental active mechanism must be used.
2. A user that holds a privilege can easily grant the privilege to another user by giving him the corresponding key. For example, if a user has a key that authorizes him to access top secret information, the user can give that key to anyone he chooses.
3. The ability to revoke a privilege that has been granted to a user requires a supplemental active protection.
4. User names must be translated into authentic keys.
5. If a client makes extensive use of key-pairs, the cost of the underlying public-key cryptography may be prohibitive.

However, cryptographic sealing provides a number of advantages not found in traditional systems:

1. Absolute privacy is provided in the sense that a user does not have to trust the computer system where his information is stored. Even a system operator or administrator can not gain access to information that he has not been authorized to see. Because there are no locksmiths for the mechanism, however, care must be taken not to lose critical keys.
2. The protection system is not privileged. Specifically, a client is free to modify its copy of the protection system code. Moreover, a client can not increase its privileges in any way by such modification.
3. Physical security of storage devices and the protection mechanism are not required. However, we assume that a client has a secure processor in which it can operate on unsealed objects.
4. It is possible to protect information that is freely distributed (e.g. a physical storage medium that is sold) and information that is broadcast.

5. Cryptographic sealing allows a simple active protection mechanism (such as the guard mechanism outlined in Section 3.2) to implement a variety of protection policies that include integrity and availability guarantees.

6. Cryptographic sealing protects variable size objects. Thus, different objects in a single file can be protected in independent ways.

These advantages make cyptographic sealing well adapted for use in decentralized computer systems.

7.6 Summary

Starting with cryptography, we described a new mechanism for protecting data. Assuming perfect cryptography, we demonstrated the secrecy and authentication properties of the mechanism. Our approach to protection is novel in that the contents of storage can be read by anyone, and yet information is kept secret from those who have not been granted access. One of the results of the mechanism is that absolute privacy is provided. A number of interesting applications of the protection mechanism were discussed, including capabilities, access control lists, and information flow control.

Exercises

1. Assume that with a known cleartext attack k can be discovered from Encrypt[z, k] and z in time T. If you are given List[Encrypt[x, k1], Encrypt[k1, k2]] and x, how long will it take you to discover *k2*?

2. How might reconfigurable and protected volumes be used together?

Chapter 8: Practical Considerations

When a complex system is designed the question of its practicality naturally arises. For a system to be practical it must supply a service that clients find useful, it must be possible to build it, and it must perform adequately. In this chapter we discuss how to realize a practical system based on the system model we have presented. The first section discusses implementation considerations, and the second section discusses configuration issues.

8.1 Implementation

Our discussion of how to implement the system model is divided into two parts. First, we will discuss prototypes of the system model and experience with these prototypes. Second, we outline a number of considerations for any full-scale implementation of the design.

8.1.1 Prototypes

The practicality of our design has been verified in part by prototypes of the transactional storage, replication, and low-level protection components of the system model. The second, third, and fourth prototypes discussed were constructed by the author.

An experimental transactional storage system called DFS [Israel et al. 1978] was constructed by a group at the Xerox Palo Alto Research Center. DFS demonstrated the feasibility of implementing transactions that span processors in a decentralized environment. DFS implements functions that are nearly identical to our description of transactional storage (Chapter 3). Although it is a prototype, the system has reached a point where it provides a useful service to many applications with acceptable performance.

The replication algorithm was implemented in an experimental system called Violet [Gifford 79a]. Violet is a decentralized calendar system that includes a simple data management system. The replication algorithm as implemented by Violet assigns one vote to each representative and fixes r and w to describe a simple majority. Experience with Violet led to the generalization of the replication algorithm described in Chapter 5, an understanding of how to structure the implementation of such an algorithm (which is shown in the code in Chapter 5), and further evidence that the services that we have proposed are useful.

The protection mechanism based on sealing (Section 7.2) was implemented in isolation. The prototype was not directly useful because it simulated the necessary cryptographic functions. The intent of the effort was to understand how to create a simple set of recursive functions to implement the protection mechanism, and to verify our understanding of the algorithm by trying out some examples on the prototype implementation. The prototype led to a simplification of our original ideas, which is reflected in the functions in Chapter 7.

Cryptographic hardware was constructed for the Xerox Dolphin processor. The hardware implements the national Data Encryption Standard [DES 75]. The hardware was designed to be fast enough that the cost of cryptography could be ignored. Encryption and decryption instructions were implemented as block transfers that included a pointer to a key in memory. Provisions were also made in the hardware for a processor instruction to generate encryption based checksums. In

addition, hardware was developed to generate true random bits that could be obtained with a processor instruction.

8.2.2 Full-Scale Implementations

We expect that a full-scale implementation of the design would adhere closely to the structure of the model implementation we have described. The object oriented programming approach we adopted simplifies the construction of a system by allowing separate ideas to be considered and implemented in isolation, with Open fitting together appropriate classes to provide aggregate services.

The details of different implementations will vary to a considerable degree. For example, there are many ways of achieving the effects of remote evaluation, such as specialized protocols [Boggs et al. 80], messages sent through pipes [Osterhout et al. 80], or a remote procedure call mechanism [Nelson 81].

Our read and write architecture for input and output does not preclude other approaches. For example, an alternate architecture for file input and output is to consider a file to logically exist in the address space of a processor [Corbato et al. 72]. This is known as mapped input and output and the read and write operations defined by our system model can be used in support of this idea. If the host operating system allows clients to allocate and directly manage portions of a processor's address space, then mapped files can be provided without modifying the host operating system. The host operating system would notify a client when a page fault occurred in a client managed address space, the client would in turn use the facilities we have described to read the appropriate page, and then the client would supply the page to the host operating system. Although no current operating system provides such facilities, they probably would not be very hard to implement.

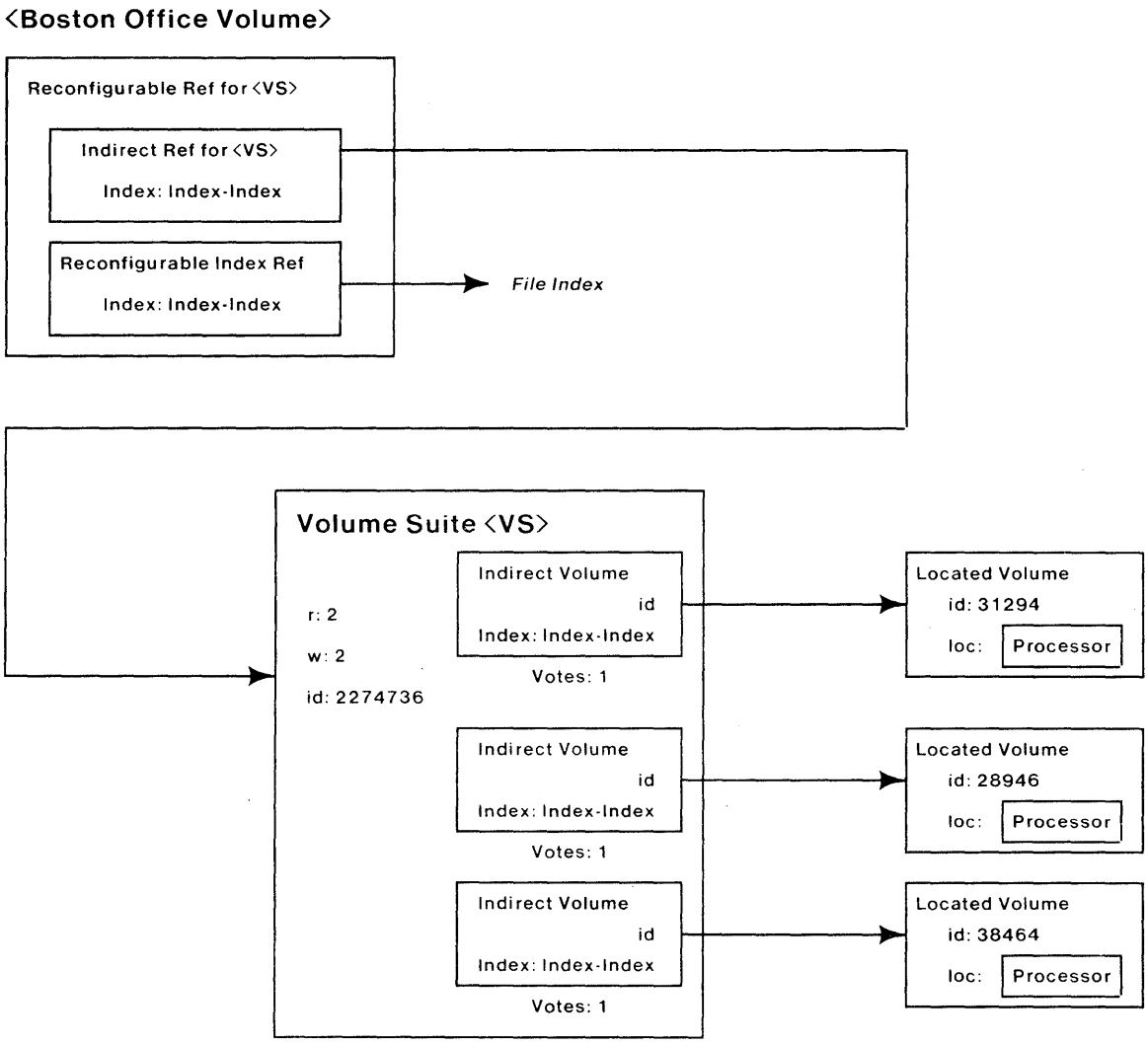
8.2 Configuration

8.2.1 Static Configuration

Once the implementation of the system has been completed, the task remains of arranging a set of physical and logical components to provide appropriate service. So far we have presented a collection of facilities without suggesting how they might be used together. Because the components we have described are fundamental, there are many ways that they can be arranged. We introduce system configuration with an example.

Figure 8.1 shows how a basic storage system might be configured. Starting with three volumes, we create indirect volumes to allow the volumes to move. The index that keeps track of the volumes' locations is the Index-Index. With the resulting three indirect volumes we create a volume suite. The volume suite is configured so that it will be available when any two of its component volumes are available. With the volume suite we create a reconfigurable volume to allow the volume suite to be reconfigured. The Index-Index is used to keep track of the storage that is used to implement the reconfigurable volume. The reconfigurable volume can be used for one of two purposes:

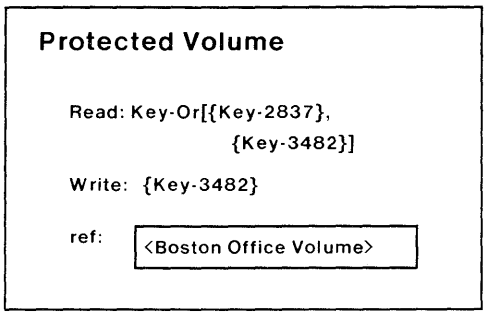
1. The entire reconfigurable volume can be reconfigured to use a different storage service. For example, the volume suite could be replaced by a single volume, or the volume suite could be reconfigured to have five volumes instead of three.



A Basic Storage Service

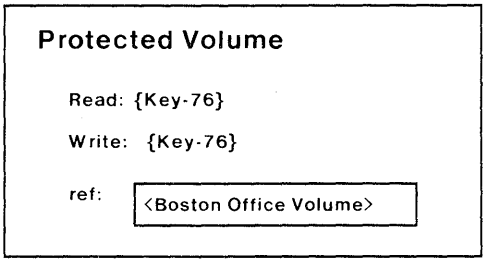
Figure 8.1

<Accounting Volume>



<Accounting Dept.>: Read
<Manufacturing>: Read
<Accounting Dept.>: Write

<John Dover's Volume>



<John Dover>: Read
<John Dover>: Write

Protection Structure

<John Dover>
{Key-76} = Seal-Only[Create-Key[]]
<Manufacturing>
{key-2837} = Seal-Only[Indirect-Key[Key-Or[<Adams> ... <Zephyr>]]]
<Accounting Dept.>
{Key-3482} = Seal-Only[Indirect-Key[Key-Or[<Adelman> ... <Wisk>]]]

Client Defined Protected Volumes

Figure 8.2

2. Individual files on the reconfigurable volume can be reconfigured to use a different storage service. For example, a file that had not been used in a long time could be moved to a low-cost storage device.

This example describes a structure that would be defined by a knowledgeable system administrator. The system administrator would configure storage systems to meet the basic requirements of his clients.

Clients can extend a basic storage service for their special needs. Figure 8.2 shows two volumes that define protected storage. Files created on <Accounting Volume> will be able to be read by users in <Accounting Dept.> or in <Manufacturing> and updated by users in <Accounting Dept.>. <Accounting Dept.> and <Manufacturing> can be changed as people join and leave the groups. This is accomplished by changing their indirect keys. Only <John Dover> can read and write files created on <John Dover's Volume>.

Note that there is no way to treat the files created on <Accounting Volume> as a group. The same is true for files created on <John Dover's Volume>. Thus, it is not possible to change the protection of all of the files created on <John Dover's Volume> because there is no way to enumerate these files. However, if <John Dover's Volume> had been a reconfigurable protected volume, then we could change the protection of all of the files that had been created on it. There is an important subtle difference between reconfigurable protected volumes and protected reconfigurable volumes.

In general, configuration is a difficult problem. This fact is hidden from naive clients, as they can use storage without knowing how it is configured.

A methodology for configuring a system is as follows:

1. First, carefully understand the needs of the client population. Clients' needs should be analyzed in enough detail to understand specific requirements for capacity, reliability, availability, performance, security, and flexibility.

2. Choose a set of hardware components that is likely to satisfy these client requirements.

3. Organize the raw capability represented by the hardware components into a set of external objects (such as volumes) that clients will use. This corresponds to specifying the client interface to the storage service without deciding how the storage service is implemented.

4. Determine how the external objects will be configured. Suite configurations, the location of volumes, an indexing structure, and so on must be selected based, in part, on the anticipated applications of the system.

   For example, a wide variety of indexing structures can be created. For a very small configuration all objects could be indexed by the index-index; for large configurations it is likely that several levels of indexes will be common (index B indexes volume C, index A indexes index B, the index-index indexes index A).

5. Review the properties of the resulting system, and compare them with the requirements established in Step 1.

The last four steps are repeated until an acceptable configuration is chosen.

It is likely that a program could help in this process. A program could have knowledge of the

types of hardware components available, possible component interconnections, and how the facilities we have proposed can be used to create desired properties. For example, the program could suggest alternative hardware and suite configurations to achieve a required level of availability.

Furthermore, when a new type of storage service is required it is important to consider how existing storage facilities might be utilized. A configuration program could maintain a data base of available storage services and their properties and automatically consider how they might be used.

In summary, capacity can be added to a configuration by adding storage devices, and the properties of storage can be improved by using the facilities described by the system model.

8.2.2 Dynamic Configuration

The configuration decisions we have just outlined so far have two properties. First, the decisions are reviewed infrequently. Second, the decisions are large in the sense that they affect the abstractions that clients see and they involve trucks dropping off new equipment at loading docks. We will call the problem of making such decisions *static configuration.*

It is also possible to take discretionary actions to improve the performance or lower the cost of a system. These decisions do not affect the objects that users see and they are essentially reviewed continuously. We will call the problem of making such decisions *dynamic configuration.* Dynamic configuration includes ideas such as:

> Using write-once storage to store a file and, when the file is opened for update, temporarily reconfiguring the file to read-write storage. This allows lower cost storage technologies to be used without the knowledge of clients.
>
> Dynamically creating weak representatives in response to perceived local needs for information.
>
> Dynamically changing suite configurations to match changes in the referencing characteristics of clients.

It is likely that decision analysis [Raiffa 70] can be applied to help formulate strategy for dynamic configuration. Decision analysis allows strategy to be formulated with incomplete information and allows the value of additional information to be judged.

8.3 Summary

We argued that the system model is practical, based on prototypes that have been constructed. The model implementation has benefited from experience with a number of prototypes. Thus, it is likely that full-scale implementations of the system model will be patterned after the model implementation. System configuration was observed to have two independent aspects. The first was the static arrangement of hardware components and logical objects to provide a useful service. The second was the problem of real-time system management which involves strategies for taking discretionary actions to improve the performance of the system.

Chapter 9: Summary of Ideas

In this paper we have considered the problem of information storage in a decentralized computer system. The major ideas that we adopted were:

1. Transactional Storage Transactions that guarantee totality, serial consistency, and external consistency were used to simplify parts of the system. As we have pointed out, all of the properties of transactions are not always required, but in some instances they provide a foundation that simplifies system design to a large degree.

2. Object Style The system model was constructed using an object oriented programming style. This style allowed a diverse set of ideas to be considered and explained separately.

From these two starting points, we introduced the following major novel ideas:

1. Naming References were introduced to name objects. Internally, a reference is a typed record. To a client a reference appears to be a variable length byte string. References can include such things as location information, protection guards, cryptographic keys, and other references. In addition, references can be made indirect to delay their binding to a specific object or location.

2. Location A new location mechanism was presented that hides the location of objects. Location was implemented by using indirection to delay the binding of references to object storage sites.

3. Replication A new replication algorithm was introduced that can improve the availability, reliability, and performance of objects. It was shown how previous replication algorithms were special cases of the new algorithm, and how temporary copies naturally fit into its framework.

4. Reconfiguration A new reconfiguration mechanism was presented that will dynamically reorganize objects. Reconfiguration is accomplished by indirection and state transfer.

5. Cryptographic Sealing A new very flexible low-level protection mechanism based on sealing objects with cryptographic techniques was introduced. The mechanism can be used for fine grained protection. Assuming perfect cryptography, the mechanism was shown to be correct.

6. Object Protection The low-level protection mechanism was used to create popular protection structures. Access control lists, information flow control, capabilities, secure communication channels, and revocation were implemented in terms of our new low-level protection mechanism.

CHAPTER 9: SUMMARY OF IDEAS

In Chapter 1 we established a set of architectural principles that the system model was to observe. Here we briefly review how the system model satisfies each of the twelve principles.

1. The system we described will always function in a well defined manner. The semantics of concurrent access to storage are well defined, and the mechanism provided for object location is always guaranteed to find an on-line object.

2. The storage service we provide is based on the ideas of files and volumes.

3. Stable storage is resilient to a set of expected failures. Furthermore, it will detect most unexpected failures.

4. References provide unambiguous low-level names that can be used in a variety of ways by clients.

5. Transactions mediate concurrent access to storage in a well defined way.

6. The assumed environment of the system model is decentralized, and a mechanism is provided for locating objects. Clients can also choose to use located references.

7. There is no inherent limit to the size of the system. Processors and storage devices can be added to increase the capacity of the system, and the number of users that it can service.

8. Suites provide a comprehensive facility for replicating objects to improve their availability, reliability, and performance. Weak representatives allow cache copies to be handled in a natural way.

9. The reconfiguration mechanism allows the storage that is used to implement a storage system object to be dynamically changed.

10. A low-level protection mechanism based on cryptographic sealing is provided, and we showed that it is secure. This mechanism is used to create popular protection policies.

11. Volume suites, reconfigurable volumes, and protected volumes allow complex configuration structures to be hidden from clients.

12. A client specifies the volumes that are used to store its information. Thus, a client can guarantee its autonomy if it so desires.

This paper has considered problems that occur in large scale information systems. We have taken care to formulate our solutions to these problems in a general way. Some of our solutions, such as the protection mechanism, will no doubt find use in a variety of applications.

In the years to come it will be a challenge to design and build large information systems. These systems will alter our life style, influence the way that we interact with each other, and even contain sensitive information about us. It is our hope that the ideas we presented will be useful to the future designers of such systems.

Appendix A: Exposition Language - EL

EL, the programming language that is used in the paper to describe algorithms, is a simple extension of Lisp 1.5. The purpose of this appendix is to fully describe the extensions we have made. The reader unfamiliar with Lisp will find the *Lisp 1.5 Manual* [McCarthy et al. 62] helpful.

A.1 Language Extensions

To improve the readability of source programs a number of minor syntactic extensions are added to M-Expressions. These extensions have been chosen to make EL programs easy to read for people that are familiar with contemporary algebraic languages.

Things that are ignored:

Any line that starts with two dashes "--" is treated as a comment and is ignored.

The expression after a colon ":" is ignored. This is so the types of variables can be included as comments in the source text. "Any" is used to indicate that a variable may contain any expression.

Transformations:

Commas are used to separate variables.

Function definitions can include a formal return variable after a slash "/". F[x/y] ← z is transformed to F[x] ← Prog[[y]; Return[z]].

The following constructs are adopted from Interlisp [Teitelman et al. 78]: IF THEN, IF THEN ELSE, WHILE DO, UNTIL DO, FOR DO, and DO. As in Interlisp, they are transformed into progs.

As in Interlisp, certain infix operators are transformed to appropriate functions and predicates. The infix predicates transformed are: =, <, >, <=, >=, # (not equal). The infix functions transformed are ←, +, -, *, /, ↑, and they must have a space on either side.

A shorthand for class invocations is provided. x | y[z] is transformed to Apply[x, List['y, z]] (see Section A.2).

Record accesses are transformed to make programs easier to read. The form x.y ← z is transformed to Store[x, 'y, z]. The form x.y without a trailing assignment operator is transformed to Fetch[x, 'y] (see Section A.3).

If a function is evaluated with too many arguments, the extra arguments are ignored. If a function is evaluated with too few arguments, NIL is supplied for the missing arguments.

A.2 Classes

Seemingly identical requests can require substantially different amounts of processing. Consider reading a file. Reading one file might not present any unusual complications, but reading another

file might require decryption, logic to determine which copy of the file to actually read, and communication to a remote processor. Ideally, a client should not be able to tell the difference between the two files.

The notion of a *class* allows algorithms to be composed with one other to provide a composite service that has a standard interface. A class can use private state and other classes to process the requests that it receives. For example, a class that implemented a replicated file might need state to keep track of what copies are current, and might use other services to read and write file copies. The concept of a class is further elucidated in [Ingalls 78].

To request that a class perform a function the class is applied to a request list. The car of the request list is the function to be performed, and the remainder of the request list consists of arguments to the function. Section A.1 introduced the shorthand for invoking a class

```
class | operation [arg1 .. argn]
```

which is transformed to

```
Apply[class, List['operation, arg1, ... , argn]].
```

When a class starts executing it has at hand a request list, and it also has private state variables that were declared when the class was created. These variables are implemented by the closure mechanism in Lisp 1.5.

Create-Class[function-map: List, superclass: Class / class: Class]

> Create-Class creates a class that can process the functions included in *function-map*. *function-map* is a list of operation names and functions to implement them. For example, if Create-Class was applied to List['Read, 'FRead, 'Write, 'FWrite] it would create a class that would evaluate FRead when it received a Read request, and FWrite when it received a Write request. If an operation is not contained in *function-map* then the request is passed to *superclass*, if it has been specified. If *superclass* is NIL and the operation is not contained in *function-map* then Error['NoFunction] is returned.

Create-Class saves the environment in which it was evaluated. This saved environment is restored whenever the class it created is evaluated. This allows a class to maintain private state between invocations. Using our previous example, FRead and FWrite would both execute in the environment that Create-Class had, and thus could save state and communicate with each other.

The following distinguished variables are defined when a class is executing:

self	The current class.
superclass	This class' superclass.
request	The current request.

The class mechanism is implemented by the following functions:

```
Create-Class[function-map: List, superclass: Class / self: Class] ← Prog[ []:
        self ← Function[Standard-Class]:
        -- self, superclass, and request will be available to the class
        Return[self]:
        ]:
```

```
Standard-Class[request: List / result: Any] ← Prog[
        [f: Function; operation: Atom; self: Class];
        operation ← car[request];
        f ← Listget[function-map, operation];
        IF Null[f] THEN [
                -- No function.  See if we have a superclass.
                IF Null[superclass] THEN Return[Error['NoFunction]];
                Return[Apply[superclass, request]];
                ];
        Return[Apply[f, cdr[request]]];
        ];

Listget[x: List, y: Atom / result: Any] ← Prog[ [];
        -- takes a list of the form [atom, val, ... atom, val]
        -- returns the value associated with y
        IF Null[x] THEN Return[NIL];
        IF car[x]=y THEN Return[cadr[x]];
        Return[Listget[cddr[x], y]];
        ];
```

A.3 Records

To make data structures easier to understand and implement the notion of a *record* is introduced. A record instance consists of a set of independent fields that can be read and written with Fetch and Store, respectively. A field is just like a variable, and is named by a tag. A record instance is made by applying Create to a record type. A record type is created by applying the function Record to a list of tags the new record might contain. It is also possible to create a new record type by combining two existing record types with Extend.

Every record is assigned a unique type. Different types of records are used for different things, and often it is convenient to be able to determine the type of a record instance. A client can use the function Is to test the type of a record instance.

Record[tag1: Atom, tag2: Atom, ... tagn: Atom / type: Record-Type]

Record creates a new record with a distinct type. The arguments to Record are discarded.

```
Record[/type: Record-Type] ← Prog[ []
        Return[List[GetUniqueID[]]]
        ];
```

Create[type: Record-Type / instance: Record-Instance]

Create creates an instance of a record. The instance inherits the types of its template.

```
Create[type: Record-Type / instance: Record-Instance] ← Prog[ []
        Return[List[Cons['type, type]]]
        ];
```

Store[instance: Record-Instance, tag: Atom, value: Any]

Store sets the value of the field named by tag to be value. Store also returns value.

```
Store[instance: Record-Instance, tag: Atom, value: Any] ← Prog[ []
        PutAssoc[tag, value, instance];
        Return[value];
        ];
```

Fetch[instance: Record-Instance, tag: Atom / value: Any]

Fetch returns the value of the field named by tag in instance. If no value has been stored in the field, NIL is returned.

```
Fetch[instance: Record-Instance, tag: Atom / value: Any] ← Prog[ []
        value ← Assoc[tag, instance];
        IF Null[value] THEN Return[NIL];
        Return[cdr[value]];
        ];
```

Extend[typea: Record-Type, typeb: Record-Type / type: Record-Type]

Extend creates a record type that is the union of its input record types.

```
Extend[typea, typeb / type] ← Prog[ []
        Return[Append[typea, typeb]];
        ];
```

Is[x: Record-Instance, y: Record-Type / result: Boolean]

The type of a record instance can be checked with Is. Is returns T if the the types of y are a subset of the types of x. Otherwise it returns NIL.

To define Is we need an auxiliary function, Subset. Subset[x: List, y: List] returns T if x is a subset of y.

```
Is[x: Record-Instance, y: Record-Type / result: Boolean] ← Prog[ []
        IF Null[x] THEN Return[NIL];
        IF Atom[x] THEN Return[NIL];
        Return[Subset[y, Fetch[x, 'type]]];
        ];

Subset[x: List, y: List / result: Boolean] ← Prog[ []
        IF Null[x] THEN Return[T];
        Return[And[Member[Car[x], y], Subset[Cdr[x], y]]];
        ];
```

Add-Type[x: Record-Instance, t: Record-Type / z: Record-Instance]

Add-Type adds the type t to the record instance specified by x. x is also returned.

```
Add-Type[x: Record-Instance, t: Type / z: Record-Instance] ← Prog[ [];
        x.type ← Union[x.type, t];
        Return[x];
        ];
```

Remove-Type[x: Record-Instance, t: Record-Type / z: Record-Instance]

Remove-Type removes the type t from the record instance specified by x. x is also returned.

```
Remove-Type[x: Record-Instance, t: Type / z: Record-Instance] ← Prog[ [];
        Efface[t, x.type];
        Return[x];
        ];
```

Examples of record usage:

```
Experienced ← Record[experience: Integer];
Person ← Record[age: Integer];
Experienced-Person ← Extend[Person, Experienced];
joe ← Create[Experienced-Person];
-- Is[joe Experienced] = T
joe.experience ← 10;
joe.age ← 26;
```

A.4 External Representation

Encode[object: Any / encoded: Byte-Array]

Encode converts an object into an array of bytes, and is similar to the Lisp function Print. An encoded object may be preserved outside of EL for any length of time. None of the data structures in the paper are circular, and thus we do not require that Encode work properly on circular structures.

Decode[encoded: Byte-Array / object: Any]

Decode converts an encoded object back into a form that can be directly manipulated. Decode is similar to the Lisp function Read. Decode[Encode[x]] is equivalent to Copy[x].

A.5 Exception Handling

```
Error-Type ← Record[problem: Any];
```

Error[problem: Any]

When an exceptional condition occurs in an EL function, the function returns what amounts to an error code. Error creates an instance of Error-Type. The argument to Error is used to set the new instance's problem field so different types of errors can be distinguished. Is[x, Error-Type] will return T if x is an error instance.

```
Error[problem / instance] ← [ [];
        instance ← Create[Error-Type];
        instance.problem ← problem;
        Return[instance];
        ];
```

A.6 Processes

Fork[form: Any / handle: Process-Handle]

Fork starts another process that evaluates the supplied form in the current environment. Fork returns a process handle that can be used with Join and Stop.

Join[handle: Process-Handle / result: Any]

Join returns the result of the evaluation carried out by the process specified by handle. If the process specified by handle has not completed its assigned evaluation Join will pause until the result is available. If no copies of a process' handle are kept then the result of the process' evaluation will be discarded.

Stop[handle: Process-Handle]

Stop causes the process specified by handle to be stopped and destroyed.

A.7 Synchronization

Create-Lock[/lock: Lock]

Create-Lock creates a lock.

Critical[lock: Lock, form: Any / result: Any]

Critical locks the specified lock, evaluates form, and then unlocks the specified lock. Critical returns the result of the evaluation of form.

Create-Condition-Variable[time-out: Time / cv: Condition-Variable]

Create-Condition-Variable creates a condition variable. If time-out is not NIL, then it is the maximum time that Wait will pause on cv.

Wait[cv: Condition-Variable]

Wait pauses until a Broadcast on the specified condition variable occurs, or the time-out expires.

Broadcast[cv: Condition-Variable]

Broadcast wakes up all processes waiting on cv.

A.8 Sets

Set-Create[/set: Set]:

Set-Create creates a new volatile set. A volatile set is a set that is destroyed when there are no more references to it or the machine it is stored on fails.

Set-Insert[set: Set, key: Any, value: Any]

Inserts or replaces a reference to *value* in *set* indexed by *key*.

Set-Lookup[set: Set, key: Any / value: Any]

Returns the value indexed by *key* in *set*, or NIL if there is no such key.

Set-Delete[set: Set, key: Any]

Deletes the entry indexed by *key* in *set*.

A.9 Byte Arrays

Byte-Array[length: Integer / array: Byte-Array]

The Byte-Array function creates an array of bytes that has a specified length. A byte is an eight bit integer. All of the elements of the array are initially set to zero.

Length[array: Byte-Array / length: Integer]

Length returns the number of of bytes in an array.

Elt[array: Byte-Array, element: Integer / value: Byte]

Elt returns the value of the specified element of the array.

SElt[array: Byte-Array, element: Integer, value: Byte]

SElt sets the value of the specified array element.

A.10 Miscellaneous

Intersection[lista: List, listb: List / result: List]

Intersection returns the list of elements that are in *lista* and *listb*.

Nth[l: List, n: Integer / element: Any]

Nth returns the nth element of list l.

Pair[lista: List, listb: List / result: List]

If *lista* is [a b c] and *listb* is [1 2 3], the result of Pair[lista, listb] is [[a 1] [b 2] [c 3]].

Union[lista: List, listb: List / result: List]

Union returns the list of elements that are either in *lista* or *listb*.

Appendix B: Storage System Summary

This appendix is a concise summary of the operations that are implemented by the storage system. The reader is referred to the text for full descriptions of these operations, and for details on the protection mechanism.

The operations we introduced to create references that are derived from other references are:

Create-Capability[ref: Reference, pl: List[Key] / cref: Capability]

Creates a capability for ref with privileges pl.

Create-Choice[choices: List[Reference] / cr: Choice]

Creates a reference whose referent can be any one of the elements in choices.

Create-Encrypted[ref: Reference, k: Key / eref: Encrypted]

Encrypts an object with k.

Create-Index[storage: File / iref: Index]

Creates an index.

Create-Indirect[ref: Reference, index: Index, tc: TC / iref: Indirect]

Binds iref to ref, but this binding can be changed with ChangeReference.

Create-Located[ref: Reference, loc: Location / lref: Located]

When lref is opened, control will be transferred to loc and ref will be opened.

Create-Protected-Volume[ref: Reference, tl: List[Type], kl: List[Key] / gref: Guarded]

Files created on a protected volume have protection controls set according to tl and gl.

Create-Reconfigurable[ref: Reference, index: Index, tc: TC / rr: Reconfigurable]

Creates a reconfigurable object, whose first implementor is ref.

Create-Revocable[ref: Reference, pl: List[Key], key-list: List[Key], ck: Key, i: Index, sp: Secure-Processor, tc: TC / rref: Revocable]

Creates a capability that may be revoked.

Create-Suite[name: UniqueID, r: Integer, w: Integer, rep: List[Reference], votes: List[Reference] / sref: Suite]

Creates a suite with the specified configuration.

Detailed below are the operations that are supported by each type of object. The bold face entries correspond to object types. The list begins with the operations that are supported by all objects, and then describes specialized operations.

Class

Close[]

Deactivates a class.

CopyReference[counted: Boolean / copy: Reference]

Copies a reference.

GetID[/ id: UniqueID]

Returns the unique identifier of the object.

GetTransactionClass[/ tc: TC]

Returns the transaction class associated with this open object.

DestroyReference[/ destroyed: Boolean]

Destroys this reference and returns T if its reference count went to zero.

Indirect

ChangeReference[new-ref: Reference]

Changes the indirect reference to point at new-ref.

Processor, Secure Processor

Eval[form: Any / result: Any]

Evaluates form at the processor serviced by this class.

Coordinator

Create-Transaction[/ tref: Transaction]

Creates a transaction.

Transaction

Abort[]

Aborts the transaction.

Commit[]

Commits the transaction.

GetStatus[/state: Atom]

Returns the current state of the transaction.

AddParticipant[commit: Global-Function, abort: Global-Function / id: UniqueID]

Adds a participant to transaction processing.

DeleteParticipant[id: UniqueID]

Deletes a participant from transaction processing.

Information Holding Object

IsImmutable[/ immutable: Boolean]

Returns T if the object is immutable.

SetImmutable[]

Makes an object immutable.

GetVersion[/ version: Integer]

Returns the object's version number.

Copy[from: Reference]

Overwrites the object with the contents of from.

Volume, Volume Suite

Create-File[name: UniqueID, exists: Boolean / ref: File]

Creates a new file.

File, File Suite

Read[startPage: Integer, pages: Integer / data: Byte-Array]

Reads from a file starting at startPage the number of pages specified by pages.

Write[startPage: Integer, pages: Integer, data: Byte-Array]

Updates a file starting at startPage by number of pages specified by pages.

GetSize[/ pageCount: Integer]

Returns the number of pages in the file.

SetSize[pageCount: Integer]

Sets the number of pages in the file.

Index, Index Suite

Enumerate[last: Entry / next: Entry]

Enumerates the contents of an index.

Read[entry-name: Byte-Array / value: Byte-Array]

Reads the value of an index entry.

Write[entry-name: Byte-Array, value: Byte-Array]

Updates the value of an index entry.

Appendix C: Open

The function Open contains the only centralized knowledge of the types of objects that are supported by the system. Thus, to add a new type of object to the system it is only necessary to add an appropriate line of code to Open.

The function Open-Local is defined to provide classes to service coordinators, files, indexes, volumes, or transactions that reside on the local processor.

If the location of a file or an index is not known, we look for it on the local processor. If it is not there, its location is computed. These two things could occur in parallel.

Although it is not shown in the code, Open could detect when an object was opened more than once with identical parameters and return the same class. In certain cases, such as opening a processor more than once, the performance gain would be significant.

```
Open[ref: Reference, tc: TC, ring: List[Key], guards: List[Key] / class: Class] ← Prog[ []:
    -- if passed NIL, return NIL
    IF Null[ref] THEN Return[NIL];

    -- Located, Capability, and Indirect have the highest precedence
    --    (because they alter where a request is processed, the keys
    --    it is processed with, and the referent that is used).
    IF Is[ref, Located] THEN Return[Open-Located[ref, tc, ring, guards]];
    IF Is[ref, Capability] THEN Return[Open-Capability[ref, tc, ring, guards]];
    IF Is[ref, Indirect] THEN Return[Open-Indirect[ref, tc, ring, guards]];

    -- The following can occur in any order.
    IF Is[ref, Choice] THEN Return[Open-Choice[ref, tc, ring, guards]];
    IF Is[ref, Encrypted] THEN Return[Open-Encrypted[ref, tc, ring, guards]];
    IF Is[ref, Protected] THEN Return[Open-Protected[ref, tc, ring, guards]];
    IF Is[ref, Processor] THEN Return[Open-Processor[ref, NIL, NIL, guards]];
    IF Is[ref, Reconfigurable] THEN Return[Open-Reconfigurable[ref, tc, ring, guards]];
    IF Is[ref, Secure-Door] THEN Return[Open-Secure-Door[ref, NIL, NIL, guards]];
    IF Is[ref, Secure-Processor] THEN Return[Open-Secure-Processor[ref, NIL, NIL, guards]];
    IF Is[ref, Suite] THEN Return[Open-Suite[ref, tc, ring, guards]];

    -- We don't know where the object is.  Try this processor.
    class ← Open-Local[ref, tc, ring, guards];
    -- if success, return to client
    IF class#Error['NotFound] THEN Return[class];
    -- Find object
    IF Is[ref, File] THEN Return[Open-File[ref, tc, ring, guards]];
    IF Is[ref, Index] THEN Return[Open-Index[ref, tc, ring, guards]];
    -- Not a File or an Index  (should not occur)
    Return[Error['NotFound]];
    ];
```

Bibliography

[Alsberg et al. 76] Alsberg, P., Belford, G. G., Day, J. D., Grapa, E., Multi-Copy Resiliency Techniques, CAC Document Number 202, Center for Advanced Computation, Univ. of Illinois, Urbana, Illinois, May 1976.

[Blakley 79] Blakley, G. R., Safeguarding Cryptographic Keys, *Proc. 1979 National Comp. Conf.*, AFIPS Press, pp. 313-317.

[Boggs et al. 80] Boggs, D.R., Shoch, J.F., Taft, E.A., Pup: An Internetwork Architecture, *I.E.E.E. Trans. on Comm. COM-28*, 4 (April 1980), pp. 612-624.

[Belady and Lehman 77] Belady, L. A., and Lehman, M. M., The Characteristics of Large Systems, Report RC-6785, IBM T. J. Watson Research Center, September 1977.

[Birrell and Needham 80] Birrell, A. D., and Needham, R. M., A Universal File Server, *IEEE Trans. on Software Engineering SE-6*, 5 (September 1980), pp. 450-453.

[Brooks 75] Brooks, F. P., *The Mythical Man-Month*, Addison-Wesley, Reading, MA, 1975.

[Chaum and Fabry 78] Chaum, D. and Fabry, R., Implementing Capability-Based Protection Using Encryption, Rep. UCB/ERL M78/46, Electronics Research Laboratory, University of CA, Berkeley, July 1978.

[Clark et al. 78] Clark, D. D., Pogran, K. T., Reed, D. P., An Introduction to Local Area Networks, *Proc. of the IEEE 66*, 11 (November 1978), pp. 1497-1516.

[Corbato et al. 62] Corbato, F. J., Daggett, M. M., Daley, R. C., An Experimental Time-Sharing System, 1962 Fall Joint Computer Conference, *AFIPS Conf. Proc. 21*, pp. 335-344.

[Corbato et al. 72] Corbato, F. J., Saltzer, J. H., Clingen, C. T., Multics - The First Seven Years, 1972 Fall Joint Computer Conference, *AFIPS Conf. Proc. 40*, pp. 571-583.

[Davies 73] Davies, C. T., Recovery Semantics for a DB/DC System, *Proc. ACM National Conference*, 1973, pp. 136-141.

[Denning 76] Denning, D. E., A Lattice Model of Secure Information Flow, *Comm. ACM 19*, 5 (May 1976), pp. 236-243.

[Dennis and Van Horn 66] Dennis, J. B., and Van Horn, E. C., Programming Semantics for Multiprogrammed Computations, *Comm. ACM 9*, 3 (March 1966), pp. 143-155.

[DES 75] Notice of a Proposed Federal Information Processing Data Encryption Standard, *Federal Register*, Vol. 40, No. 52, March, 1975.

[Diffie and Hellman 76] Diffie, W., and Hellman, M. E., New Directions in Cryptography, *IEEE Trans. on Inf. Thy. IT-22*, 6 (November 1976), pp. 644-654.

[Diffie and Hellman 77] Diffie, W., and Hellman, M. E., Exhaustive Cryptanalysis of the NBS Data Encryption Standard, *Computer* (June 1977), pp. 74-84.

[Diffie and Hellman 79] Diffie, W., and Hellman, M. E., Privacy and Authentication: An Introduction to Cryptography, *Proc. of the IEEE 67*, 3 (March 1979), pp. 397-427.

[Dion 80] Dion, J., The Cambridge File Server, *Operating Systems Review 14*, 4 (October 1980), pp. 26-35.

[Eswaran et al. 76] Eswaran, K.P. et al., The Notions of Consistency and Predicate Locks in a Database System, *Comm. ACM 19*, 11 (November 1976), pp. 624-633.

[Forsdick et al. 77] Forsdick, H. C., Schantz, R. E., Thomas, R. H., Operating Systems for Computer Networks, Rep. 3614, Bolt Beranek and Newman, Cambridge, MA, July 1977.

[Garnett and Needham 80] Garnett, N.H., and Needham, R.M., An Asynchronous Garbage Collector for the Cambridge File Server, *Operating Systems Review 14*, 4 (October 1980), pp. 36-40.

[Gifford 79a] Gifford, D.K., Violet, An Experimental Decentralized System, Integrated Office System Workshop, IRIA, Rocquencourt, France, November, 1979. Available as CSL Report 79-12, Xerox Palo Alto Research Center, 1979.

[Gifford 79b] Gifford, D.K. Weighted Voting for Replicated Data, *Operating System Review 13*, 5 (December 1979), pp. 150-162.

[Gray et al. 76] Gray, J. N. et al., Granularity of Locks and Degrees of Consistency in a Shared Data Base, in *Modeling in Data Base Management Systems*, North Holland Publishing, 1976, pp. 365-394.

[Gray 78] Gray, J. N., Notes on Data Base Operating Systems, in *Operating Systems, An Advanced Course*, Lecture Notes in Computer Science 60, Springer-Verlag, 1978, pp. 393-481.

[Gray et al. 79] Gray, J. N., et al., The Recovery Manager of a Data Management System, Rep. RJ2623, IBM San Jose Research Laboratory, San Jose, CA, August 1979.

[Gudes 80] Gudes, E., The Design of a Cryptography Based Secure File System, *IEEE Trans. on Soft. Eng. SE-6*, 5 (September 1980), pp. 411-420.

[Hellman 76 et al.] Hellman, M. E., et al., Results on an initial attempt to Cryptanalyze the NBS data encryption standard, Rep. SEL-76-042, Dept. of Electrical Engineering, Stanford Univ., Stanford, CA, September 1976.

[IBM 80a] *CICS/VS System Application Design Guide*, Third Edition, Rep. SC33-0068-2, International Business Machines, 1980.

[IBM 80b] *IBM 3850 Mass Storage System Introduction and Preinstallation Planning*, Second Edition, Rep. GA32-0038-1, International Business Machines, 1980.

[Ingalls 78] Ingalls, D. H., The Smalltalk-76 Programming System Design and Implementation, *Fifth Annual ACM Symposium on Principles of Programming Languages*, Tucson, Arizona, January 1978.

[Israel et al. 1978] Israel, J.E., Mitchell, J.G., and Sturgis, H.E., Separating Data From Function in a Distributed File System, Report CSL-78-5, Xerox Palo Alto Research Center, Palo Alto, CA, September 1978.

[Knuth 73] Knuth, D. E., *The Art of Computer Programming*, Vol. 3, *Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[Lamport 78a] Lamport, L., The Implementation of Reliable Distributed Multiprocess Systems, *Computer Networks 2* (1978), pp. 95-114.

[Lamport 78b] Lamport, L., Time, Clocks, and the Ordering of Events in a Distributed System, *Comm. ACM 21*, 7 (July 1978), pp. 558-565.

[Lampson 69] Lampson, B. W., Dynamic Protection Structures, 1969 Fall Joint Computer Conference, *AFIPS Conf. Proc. 35*, pp. 27-38.

[Lampson 80] Lampson, B. W., Replicated Commit, Draft Paper, Xerox Palo Alto Research Center, Palo Alto, CA, November 1980.

[Lampson and Sproull 79] Lampson, B. W., and Sproull, R. F., An Open Operating System for a Single-User Machine, *Operating System Review 13*, 5 (December 1979), pp. 98 - 105.

[Lampson and Sturgis 79] Lampson, B. W., and Sturgis, H. E., Crash Recovery in a Distributed Data Storage System, Draft Report, Xerox Palo Alto Reserch Center, Palo Alto, CA, April 1979. Also to appear *Comm. ACM*.

[LeLann 81] Le Lann, G., A Distributed System for Real-Time Transaction Processing, Fourteenth Hawaii International Conference on System Science, Honolulu, January 1981.

[Liddle 76] Liddle, D., private communication.

[Lindsay and Gligor 78] Lindsay, B., and Gligor, V., Migration and Authentication of Protected Objects, *IEEE Trans. Soft. Eng. SE-5*, 6 (November 1979), pp. 607-611.

[McCarthy et al. 62] McCarthy, J., et. al., *Lisp 1.5 Programmer's Manual*, MIT Press, Cambridge, MA, 1962.

[McCreight 77] McCreight, E. M., Pagination of B*-Trees with Variable-Length Records, *Comm. ACM 20*, 9 (September 1977), pp. 670-674.

[Metcalfe 73] Metcalfe, R.M., *Packet Communication*, Ph.D. Thesis, Harvard University, December, 1973. Also available as Rep. TR-114, MIT Laboratory for Computer Science, Cambridge, MA.

[Metcalfe and Boggs 76] Metcalfe, R.M., and Boggs, D., Ethernet: Packet Switching for Local Computer Networks, *Comm. ACM 19*, 7 (July 1976), pp. 395-403.

[Morris 73] Morris, J. H., Protection in Programming Languages, *Comm. ACM 16*, 1 (January 1973), pp. 15-21.

[NBS 80] National Bureau of Standards, DES Modes of Operation, Preliminary Copy of Federal Information Standards Publication 81, September, 1980.

[Needham and Schroeder 78] Needham, R. M., and Schroeder, M. D., Using Encryption for Authentication in Large Networks of Computers, *Comm. ACM 21*, 12 (December 1978), pp. 993-999.

[Needham 79] Needham, R. M., Adding Capability Access to Conventional File Servers, *Operating Systems Review 13*, 1 (January 1979), pp. 3-4.

[Nelson 81] Nelson, B., *Remote Procedure Call*, Ph.D. Thesis, Carnegie-Mellon Univ., May, 1981.

[Obermarck 80] Obermarck, R., Global Deadlock Detection Algorithm, Rep. RJ2845, IBM San Jose Research Laboratory, San Jose, CA, August, 1979.

[Osterhout et al. 80] Osterhout, J. K., Schelza, D. A., Sindhu, P. S., Medusa: An Experiment in Distributed Operating System Structure, *Comm. ACM 23*, 2 (February 1980), pp. 92-105.

[Peterson and Weldon 72] Peterson, W. W., Weldon, E. J., *Error-Correcting Codes*, Second Edition, MIT Press, Cambridge, MA, 1972.

[Popek et al. 81] Popek, G., et al., LOCUS: A Network Transparent High Reliability Distributed System, *Operating Systems Review 15*, 5 (December 1981), pp. 169-177.

[Raiffa 70] Raiffa, H., *Decision Analysis: Introductory Lectures on Choices under Uncertainty*, Addison-Wesley, Reading, MA, 1975.

[Redell 74] Redell, D. D., Naming and Protection in Extendible Operating Systems, Ph.D. Thesis, University of California, Berkeley, September 1974. Available as TR-140, MIT Laboratory for Computer Science, Cambridge, MA.

[Reed 78] Reed, D. P., Naming and Synchronization in a Decentralized Computer System, Rep. TR-205, MIT Laboratory for Computer Science, Cambridge, MA, 1978.

[Rivest et al. 78] Rivest, R. L., Shamir, A., and Adleman, L., A Method of Obtaining Digital Signatures and Public-Key Cryptosystems, *Comm. ACM 21*, 2 (February 1978), pp. 120-126.

[Rothnie et al. 77] Rothnie, J. B., Goodman, N., Bernstein, P. A., The Redundant Update Methodology of SDD-1: A System for Distributed Databases (The Fully Redundant Case), Technical Report CCA-77-02, Computer Corporation of America, Cambirdge, MA, June 1977.

[Saltzer and Schroeder 79] Saltzer, J. H., and Schroeder, M. D., The Protection of Information in Computer Systems, *Proc. of the IEEE 63*, 9 (September, 1975), pp. 1278-1308.

[Shamir 79] Shamir, A., How to Share a Secret, *Comm. ACM 22*, 11 (November, 1979), pp. 612-613.

[Shannon 49] Shannon, C. E., Communication Theory of Secrecy Systems, *Bell Sys. Tech. Journal 28*, (October 1949), pp. 656-715.

[Spector 80] Spector, A. Z., Performing Remote Operations Efficiently on a Local Computer Network, Stanford Department of Computer Science Report STAN-CS-80-850, Stanford Univ., Stanford, CA, December 1980.

[Swinehart et al. 79] Swinehart, D., McDaniel, G., Boggs, D., WFS: A Simple Shared File System for a Distributed Environment, *Operating System Review 13*, 5 (December 1979), pp. 9 - 17.

[Tandem 81] *Data Base Management ENCOMPASS: TMF*, Tandem Computers, Cupertino, CA, 1981.

[Teitelman et al. 78] Teitelman, W., et al., *Interlisp Reference Manual*, Xerox Palo Alto Research Center, Palo Alto, CA, 1978.

[Thacker et al. 79] Thacker, C. P. et al., Alto: A Personal Computer, Rep. CSL-79-11, Xerox Palo Alto Research Center, Palo Alto, CA, August 1979.

[Thomas 73] Thomas, R. H., A Resource Sharing Executive for the ARPANET, 1973 National Computer Conference, *AFIPS Conf. Proc. 44*, pp. 155-163.

[Thomas 79] Thomas, R. H., A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases, *ACM Trans. on Database Systems 4*, 2 (June 1979) pp. 181-209.

[von Neumann 51] von Neumann, J., Various Techniques Used in Connection with Random Digits, in *Monte Carlo Math Series #12*, National Bureau of Standards, 1951.