INFORMATION TO USERS

This reproduction was made from a copy of a manuscript sent to us for publication and microfilming. While the most advanced technology has been used to photograph and reproduce this manuscript, the quality of the reproduction is heavily dependent upon the quality of the material submitted. Pages in any manuscript may have indistinct print. In all cases the best available copy has been filmed.

The following explanation of techniques is provided to help clarify notations which may appear on this reproduction.

1. Manuscripts may not always be complete. When it is not possible to obtain missing pages, a note appears to indicate this.

2. When copyrighted materials are removed from the manuscript, a note appears to indicate this.

3. Oversize materials (maps, drawings, and charts) are photographed by sectioning the original, beginning at the upper left hand corner and continuing from left to right in equal sections with small overlaps. Each oversize page is also filmed as one exposure and is available, for an additional charge, as a standard 35mm slide or in black and white paper format.*

4. Most photographs reproduce acceptably on positive microfilm or microfiche but lack clarity on xerographic copies made from the microfilm. For an additional charge, all photographs are available in black and white standard 35mm slide format.*

***For more information about black and white slides or enlarged paper reproductions, please contact the Dissertations Customer Services Department.**

Fowler, Robert Joseph

DECENTRALIZED OBJECT FINDING USING FORWARDING ADDRESSES

University of Washington PH.D. 1985

Decentralized Object Finding Using Forwarding Addresses

by

Robert Joseph Fowler

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

Doctor of Philosophy

University of Washington

1985

Approved by Richard F Jadrn
(Chairperson of Supervisory Committee)

Program Authorized
to Offer Degree Department of Computer Science

Date December 4 1985

Doctoral Dissertation

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that copying of this thesis is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to University Microfilms, 300 North Zeeb Road, Ann Arbor, Michigan 48106, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature Robert Joseph Fowler

Date 4 December 1985

University of Washington

Abstract

Decentralized Object Finding Using Forwarding Addresses

by Robert Joseph Fowler

Chairperson of the Supervisory Committee: Professor Richard Ladner

Department of Computer Science

A problem that must be solved in designing a large distributed computing system is ensuring that processors can "find" all of the "objects" that they have the right to access. To avoid the potential reliability problems and bottlenecks that can be introduced by centralized services it is desirable that object finding be decentralized and allow any group of processors to proceed independently and autonomously. Given a model of a decentralized system in which *objects* are analagous to moveable physical objects and in which access rights are carried only by *proper names*, we examine in detail the implications of using *forwarding addresses* for finding objects. Special emphasis is placed on the issues of performance, active resource management, and high availability through distribution.

We analyze the cost of using each of several object finding protocols based on forwarding addresses, deriving ammortized worst case upper and lower bounds as well as estimates for the average case. The average case costs of two simple protocols are analyzed using Markov chains, revealing that the mean cost per access is order of the square root of N, where N is the number of distinct processors that an object visits. The worst

case costs of a path compressing protocol are determined to be order of $\log N$. If the object is accessed a times and moves m times, then all cost estimates are decreasing functions of a/m.

We provide a detailed specification of a possible implementation of the *PCacc* protocol. One extension of the basic implementation involves the addition of a mechanism called an ***inexact reference set.*** This extension enables a processor to actively manage its resources, both enabling it to reclaim the storage occupied by forwarding addresses and to control the use of storage occupied by objects. Another extension that we describe is that of allowing forwarding addresses to point to the multiple processors that host distributed objects. We show that this extension ensures enhanced accessibility in the presence of processor faults.

Table of Contents

List of Figures

ACKNOWLEDGEMENTS

There have been several major influences on this work. My interest in decentralized computing originated while designing the initial architectural specification of the Eden system at the University of Washington. Michael Fischer, my advisor at that time, inspired the idea of abstracting and analyzing object finding. Larry Ruzzo provided extensive feedback during my explorations into models of object naming in distributed systems. Finally, as my last advisor Richard Ladner continued to provide much needed criticism in addition to making substantive suggestions for the improvement of the complexity analyses. Discussions on naming issues with John Demco were very helpful and he introduced me to the philosophical literature on naming. The average case analyses using Markov chains were successful due to discussions with and suggestions from John Zahorjan. John and Hank Levy were on the reading committee and they provided useful feedback on the form of the presentation.

This work was begun while the author held an IBM Graduate Fellowship. It continued while the author was receiving support from DARPA Contract MDA-903-82-C-0424 as well as NSF Grants MCS-8004111 and DCR-8402565.

Chapter 1

Introduction.

One of the more significant developments in the area of computer systems in the last decade has been the emergence of practical computer networks and the construction of decentralized applications upon them. The same era has also witnessed the emergence of "object–oriented" programming and of systems and languages to support it. It is natural to try to extend the "object–oriented" approach to decentralized systems.

This dissertation is an investigation of a fundamental problem that must be solved in order to have large, practical decentralized "object-oriented" systems: What mechanism should we build into the system to allow any processsor in such a system to "find" all of the "objects" that it can legally name? Conversely, which objects must it be able to find? To answer these questions is to define and solve what we call the "object finding problem".

We percieved the use of *forwarding addresses* to be a natural mechanism for object finding and therefore chose to investigate in depth the implications of using them. Emphasis is placed on the issues of performance, resource management, and high availability through replication. While forwarding addresses are naturally used in a decentralized way they can also efficiently emulate centralized mechanisms such as directories. While a directory can be accessed in constant time we will show that even in the decentralized case, the worst case ammortized cost to find an object is $O(\log N)$, where N is the number of processors that the object visits in its lifetime. In addition, we extend the basic forwarding address mechanism to provide support for active resource management and

for accessing replicated objects reliably.

There are simple and efficient mechanisms for decentralized object finding for some special cases:

- If objects never move then one can use a naming system that encodes the location of an object in its name. The object finding problem reduces to the problem of finding a route to a known location. If an object is able to anchor part of itself in a fixed location while the rest of it moves around the system then finding the anchored part also finds the rest of the object.
- If the objects that move cannot be referenced remotely then the question is moot.
- If the number of processors is not very large and objects don't move very often, then a combination of caching the last known address of an object combined with an exhaustive search may be a reasonable approach.
- If the population of objects is sufficiently small and static then a global database that has a known location, or is otherwise easy to find, can be used to keep track of object locations. This can be extended somewhat to larger systems by distributing the database. Further extension is possible by partitioning the name space and delegating responsibility for each partition to a sub-database.

These approaches work under the assumptions stated. We are interested, however, in the specific object finding problem that would exist in a completely decentralized system in which there is a very large number of processors, in which each processor may be creating and destroying small objects at a high rate, in which objects may be freely moved from processor to processor, and in which there is a mechanism for remotely referencing and accessing objects.

The ability to move and and remotely access resources in a distributed system promises many potential benefits with respect to performance and availability. For this reason, if the mechanisms to economically support the movement of remotely accessible small objects are developed then we expect them to become common in decentralized

object–oriented programming environments. While we do not know for sure that this will occur it is certain that if these operations are not efficient then they will definitely not be used. Until efficient support for these operations is provided we have no way of knowing how or how much they will be used.

An example of an environment in which we expect there to be a lot of object motion is the proposed Oz system. This system is being designed to support objects with widely ranging sizes, including small, mobile ones. For instance, its language, Emerald [BHJL85, Hut85], has a "call by move" mode in which the objects that are arguments to an operation on a remote object will be moved to that object's processor so that local references can be used to access them.

Object finding is a restricted form of what has been called "name resolution". To paraphrase Saltzer [Sal78], name resolution is the process of locating an object given a name and a context. A stage of the resolution process maps a name in a context to a new name in a new context. It may take many stages. Name resolution includes the interpretation of addresses in virtual memory systems, the organization of hierarchical directory structures, the resolution of symbols by compilers and linkers, and the functioning of name servers.

Name resolution is a very broad topic. To focus our efforts of a more manageable problem we use "object finding" to refer to the abstract but specific problem of determining which processor has the representation of an object, given a symbol that can be thought of as a *proper name* for that object. Objects are constrained to be entities that behave in a way that is intuitively analagous to physical objects, as opposed to other abstractions such as properties, classes, or abstract mathematical objects. Object finding does not include other forms of name resolution such as the use of databases for resolving descriptive names into proper names, nor does it deal with the issue of name resolution within a single processor.

We use "object finding" rather than "object location" because the latter term is easily misinterpreted to refer to the problem of choosing where to locate objects.

1.1 Other Aspects of Name Resolution.

There are many aspects of name resolution which we recognize as important but which we do not address. There will be abstract entities in a system other than objects and there will be naming schemes other than our proper names. We explicitly do not deal with these issues.

We will not deal with routing messages between pairs of processors. We will assume the existence of a logical network abstraction in which a processor that learns the name of another processor can send it a message. The logical network handles the problem of finding efficient and reliable routes over which to send the message.

We do not deal with the problem of resolving names that can be passed between processors by mechanisms outside the system. In particular, we do not deal with the problem of resolving names that can be typed or otherwise entered by humans. This is the domain of conventional *name services* [Ter85]. Name services can be constructed on top of or in parallel with the model of objects and proper names.

Similarly, we do not deal with the issue of finding objects that are referenced by descriptions in the form of sets of well publicized attributes. That is the domain of distributed databases.

1.2 An Example

In order to make our presentation more concrete we will use an extended example, the Humongous Widget Corporation, a fictional manufacturing firm of the future. Humongous has implemented a company–wide, object–oriented, decentralized computer system. The Humongous Object Programming System (HOPS) emphasizes the analogy between its objects and physical entities. Objects can come in all sizes. The mobility of an object depends upon its size and function. Small ones can be transmitted in a single inter–processor message. Large ones can represent entities as large and immobile as a factory.

The company has facilities at many sites world wide and each site may be the location

of several departments from each of the corporation's many administrative units. Each of these units enjoys a degree of decentralized authority. Each unit is internally responsible for accounting for its own resources.

We will concentrate on three kinds of application that coexist on this system.

1.2.1 Automated Manufacturing.

Humongous manufactures its widgets in automated factories. In these factories the processors in the robots are connected to the network and are programmed in HOPS.

Each tool in the factory has a corresponding object that records its history. As the tool is moved from location to location its history object accompanies it. Thus, the robot using the tool will record each operation it performs with it and send it to the toolmakers' shop for maintenance when appropriate.

Humongous prides itself on its quality control. While small parts may be anonymous, complex machined parts and assemblies are each given a unique identity. As the parts and assemblies progress through the factory a record is kept of each operation and inspection as they are performed. These records are part of a *traveller*, a HOPS object that represents the part. The traveller is sent to the part's next destination when each robot completes its operation. When the robot at the destination is ready to perform the next operation it retrieves the part from a holding area. Some stations assemble parts into larger assemblies. Sometimes when this happens the individual components lose their identities and are replaced by a larger object. Sometimes the larger object is represented by an aggregation of its components.

The manufacturing operation is very large. The manufacturing control application exhibits extreme concurrency. The travellers and tool histories are updated each time an operation is performed. This makes centralized databases for these items infeasible.

Most of the time the operations on the travellers and tool histories are strictly local. Only the robot performing the current operation need know where an object is and what its state is.

Occasionally, there is a need to remotely find and access these objects. If a part fails

an inspection then it may be because a tool is misaligned. That tool and all of the parts it has operated on may need to be found for special handling. Periodically it is necessary to perform an inventory of tools and parts. Sometimes this is done for accounting purposes, but it is also necessary in order to schedule and control the manufacturing process.

1.2.2 Automated Offices.

Humongous has automated its offices. Bureaucrats no longer push paper, rather they process HOPS objects that serve as analogues of the documents that they replaced. The methods used to process these document objects are extensions of the methods used for the decentralized processing of paper documents. They are passed from workstation processor to processor and at each they are manipulated locally, either with or without human intervention. Copies can be made, filed, and circulated, but consistent with the physical analogy each copy has its own distinct identity as an object.

Some document objects are secure in that they protect themselves against unauthorized access, copying, and printing. Other documents can route themselves automatically as they collect digital signatures. Others record the identities of copies as they are made. The manufacturing division uses HOPS documents to represent engineering drawings. When a master drawing is replaced all of its copies can be found and replaced also.

In order to ensure long term reliable storage of important document objects they can be kept at central locations and can be accessed remotely using a database query language. The sheer volume of documents processed, however, necessitates that in general they be handled in a decentralized way.

For example, an employee needing to go on a business trip might begin by creating a blank "travel requisition" object. After completing a section justifying the trip she would forward the requisition to her supervisor's work station. The supervisor might approve the trip and forward the requisition to the purchasing department which would add its approval and send it on to an automated travel agent which would communicate directly with airline and hotel computers to make the necessary arrangements. The requisition then gets routed to accounting for approval and the generation of a travel advance. The

advance and a travel expense report object are sent to the employee. After the trip the expense report is completed and it is automatically routed through the workstations of supervisors, secretaries, and accountants. If all of this happens in a timely fashion then the documents are accessed only locally by each of the processors they visit. If, on the other hand, there are delays then it may be necessary to track them down. If approval for the trip is slow in coming then the employee may want to discover the source of the problem. If the final report is slow in coming then the accounting department may have to track it down.

Some documents might last a long time and others will be destroyed as soon as their useful lives are completed. The travel requisition and final report forms can be permanently filed, but a lot of the information within them is not of long term interest. It is more likely that soon after the accounting for the trip is completed that the information with long term interest will be transferred to other documents kept in database storage and the once mobile originals be destroyed.

1.2.3 Scientific Computation.

Humongous Research Laboratories is the research and development arm of the corporation. As part of its operations HRL does a lot of scientific computation. Its scientists and engineers build very large finite element models of Humongous Widgets. Their object–oriented approach to this is an extension of the object–oriented simulation methodology of Simula 67 [DMN70]. Each model is constructed hierarchically out of HOPS objects that represent the subassemblies of the widget. Ultimately each part is represented as an HOPS object and finite element models of small volumes of the part are in turn represented by HOPS objects.

The Humongous Labs operations research and industrial engineering departments also do large combinatorial optimizations and simulations in attempts to improve the operation of the corporation's automated factories.

These intensive computations are decentralized. The laboratory maintains a large pool of processors that are willing and capable of performing these computations. A

load sharing algorithm automatically moves the sub-task objects to processors that are willing to work on them.

1.3 Objects.

While there have been several "object–oriented" computer architectures proposed and built [Lev84], we make no special assumptions about the logical processors in our hypothetical system. They may be implemented by ordinary sequential processors, "object–oriented" machines, multi–processors, or even local area networks. We assume only that each processor appears to be a single site with respect to the network as a whole.

The underlying logical processors compute by manipulating data values and storing them in local memory. We will call the concrete data structures defined in terms of values and cells *data items*. On top of this level a decentralized operating system kernel together with an object–oriented programming environment implemented on top of it defines and implements various kinds of abstract entities. In particular, we assume that they define an "object" abstraction.

We intend that these objects behave in a way that allows us to use our intuition of the behavior of everyday physical objects to help us understand and manipulate them. Ordinary physical objects have an identity of their own that is more basic than any naming system we may have for them. The only way of affecting a physical object is to interact directly with it. That interaction is local. Physical objects do not interact over a distance without mediation by other entities. Ordinary physical objects appear to move continuously without dematerializing and reappearing elsewhere. Our insistence on the analogy between objects and instances of physical entities distinguishes them from other abstractions within the system and distinguishes our definition of "object" from others. Of the models of distributed "object–oriented" computing that have appeared in the literature, the closest in spirit to ours is Hewitt's Actors formalism [HBS73, HB77].

Saltzer [Sal78] defines "object" as "a software (or hardware) structure that is considered to be worthy of a distinct name". In our model there will be abstract entities other than objects and there will be methods for naming them. Many abstractions are

defined descriptively in terms of the roles they play. For example, on a Unix (TM) system the entity whose role is "the standard C compiler" is accessed using the path name "/usr/bin/cc". If the old compiler is replaced with a new one then using that path name will access the new one. Although they perform the same role at different times we do not say, however, that the new compiler is identically the same object as the old one.

Similarly, the referent of the descripion "the processor (or processors) with the best response time" is not an object. At any one time the description will refer to some set of physical objects, but over time the referent will jump discontinuously and unpredictably around the network. This abstraction cannot be an object.

In "object-oriented" programming environments such as Simula67 [DMN70] and Smalltalk [GR83] a major emphasis is placed upong the data abstraction aspect of objects. In these monolithic environments an object is an instance of an abstraction called a class and it inherits many of its properties from a chain of other entities in what is termed a class hierarchy. Changing an element in the class hierarchy instantaneously and simultaneously affects everything below it. In contrast, a physical object, once created, is changed only by direct interaction. If the creator of a class of physical object discovers an improved way of making them, then the changes do not retroactively affect instances created in the past. For instance, when General Motors upgrades their automobiles the changes do not automatically affect previous models. Changes to those items can be affected only through an expensive and explicit recall and repair process. A mechanism for such an upgrading process can be built on top of the underlying objects but it is not an inherent part of their definition.

In our model of objects as physical analogues, the only way to affect an object is to interact directly with its representation. This defines the identity of the object. Since an abstract object is represented by data items, if changing a data item can change the state of an object or conversely if changing the abstract state of an object can change the data item, then that data item is currently part of the representation of the object. These properties constitute an operational criterion for deciding what is and is not part of a particular object's representation. For example, if an object can be moved by storing

a value in a data structure such as an entry in a directory, then that entry must be part of the object's representation.

If all of the data items in an object's representation must be located at a single processor whenever that object is accessed or otherwise observed then we will call it an *atomic object* or *atom*. Objects can also be distributed. It may be that while an atom is being moved that it is represented by data items on more than one processor, but it cannot be accessed or otherwise observed in this condition. An object, perhaps distributed, whose components are themselves atoms we will call a *molecule*. A distributed object whose components are not themselves objects with their own independent identities we will call an *amoeba* [Lam78a].

1.4 A Large Decentralized System.

The major difficulty in dealing with distributed computation is that one must take into account the physical effects of distribution. Because components that are physically separated can fail independently, reliability is an issue. Because communication is not instantaneous, the effects of communication delay must be taken into account. Processors compute asynchronously and concurrently. It is in general not possible for short lived and physically separated entities to learn of each others' existence much less communicate. A processor can have at most partial information about the state of another. The models of computation used for the study of large distributed systems must be able to incorporate these phenomena.

We assume that the underlying system is very large and decentralized. At any time it is composed of a very large number of processors and a communication network. For the sake of generality we do not assume that the network is necessarily connected. As the system evolves it changes its configuration. New processors are added and old ones fail and are junked. New network conections are added and old ones break.

This system exhibits considerable parallelism. In particular, the rate at which objects are created, accessed, moved, and destroyed is directly proportional to the number of processors.

The system is "open" in the sense that it will be impossible for any one processor to interact with all of the other processors let alone all objects even within one connected component of the network. The sheer number of objects combined with the high rate at which they are created and destroyed ensures this. This means that in general computations that depend upon knowing global states of the system cannot be guaranteed to be correct. Predicates such as "no object in the system (or my connected component) has property X" where X is not trivial can be impractical or impossible to evaluate. If X is the property "can refer to object Y" and the naming system allows the unrestricted distribution of names then the use of global garbage collection to manage the storage of objects and data structures used to find them may also be impractical or impossible.

The processors will not all be owned by the same administrative organization. A corollary of this is that there will not be any single administrator. The system will be partitioned into a set of administrative domains that exhibit some degree of autonomy.

The physical resources of the system are real economic goods that in the end need to be purchased with real money. This, with the automomy of administrative domains, means that accounting or at least accountability is important. Applications that cross administrative boundaries need to define the obligations of all parties involved to the application. The administrator of a processor should be able to control the use of its resources. The use of resources should be the result of the negotiation of agreements between administrations. There should be no "open-ended" commitments. A processor needs to be able to re-negotiate the allocation of its resources. The SPICE Butler [Dan82] addresses some of these issues.

An application in this system is characterized by the set of objects that embody its processes and data. Objects can join and leave an application and they can be moved between processors. Many applications will be localized within the administrative domains. Some applications will span a small number of administrative domains. There will be other applications that will be system wide.

We expect that applications on this system will exhibit reasonable locality properties. The vast majority of objects will be small and will be created, used, and destroyed entirely

on one processor without any other processors needing to learn of their existence. There will be a smaller but still very large number of objects each of which will be shared by a small set of processors. Of these the greatest number will be created on one processor and move to only one other processor during its lifetime. For example, in the Emerald "call by move" [Hut85] an object may be created at one processor and passed in a message to another as an argument to an operation on an object at the latter. The sender and the receiver of the request are the only two processors that need to be able to refer to it, and in many cases neither processor will have reason to refer to it again.

The set of objects that a large number of processors will be able to reference and therefore need to find will be smaller still.

The mobility of an object will usually be negatively correlated with the number of processors that need to access it. We expect that important and well-known system resources tend to be large and not move around or move in a very restricted way. On the other hand, objects that move frequently, even if they visit a large number of processors, are typically be of interest only to a small set of processors over any short interval of time.

Programmers of applications on the system will have a uniform view of objects. Whether they are large or small, nomadic or sedentary, there should be a single uniform mechanism for finding and accessing them.

1.5 Proper Names.

There are several ways of denoting an object. *Ostentive reference* [Kri72] is the direct manipulation of an object's state. It includes primitive symbolism such as touching or pointing. Since computer systems are composed of a rich hierarchy of symbolic abstractions we are faced with the problem of choosing a level to regard as primitive and to regard as "directly manipulating" objects. Because we are interested in distribution we will regard as primitive all aspects of object finding within an individual processor. We define a *handle* for an object to be a data item that is part of its representation and which ostentively names (perhaps indirectly) all of the other data items in the object's

representation. The finding of an object is thus reduced to the problem of finding a handle for that object.

Symbolic reference is the use of a symbol called a *term* to denote to an object. Some terms are in the form of *descriptions* in an appropriate language. There are several difficulties with using descriptions to denote objects.

- If the language used to encode descriptions is too powerful then it may be difficult or impossible to resolve them. We wish to rule out descriptions such as "an object whose representation is an encoding of a Turing machine that does not halt when given itself as input".
- A processor may describe objects that it should not reasonably be expected to access. It may describe local objects hidden on other processors, objects which it is specifically prohibited from accessing, and objects that are spatially and temporally separated from it in such a way as to make access a practical impossibility.
- A description can refer to different objects or sets of objects at different times. The extension of a description can thus appear to move in unpredictable and discontinuous ways. In such cases the referent could not be thought of as an object. For example, consider the description "the processor with the instantaneously shortest response time".

In order to avoid these difficulties we restrict our discussion of object finding to use a restricted form of term that intuitively plays the role of a *proper name*. We require that proper names have the following properties:

- A proper name must refer to a unique individual. We call that individual its *referent*. This property has been called *univocality* [Car56]. Each individual may have many proper names.
- A proper name must be persistent. If it refers to an individual at one time then it must always refer to that individual. Scott [Sco70] makes the point that individuality is a property of an object that necessarily must persist in all possible futures. As the state of the object changes, its identity should remain invariant. Descriptive terms such as "the Chairman of the Computer Science Department" or "/usr/bin/cc" may refer to a unique individual at any time, but these are descriptions of (terms for) the abstract roles played by these individuals rather than their proper names. In time a new Chairman or an improved C compiler will be installed and these terms will then refer to the replacements. The roles denoted by these descriptions are not objects.

- For our purposes the most important aspect of proper names is that they refer to or *denote* their referents. We do not care about the *connotation* (sense, intension) of a proper name. For example, "Morning Star", "Evening Star", and "Venus" all refer to the same planet and it is important to poets and philosophers which of these is used in a particular situation. We specifically do not want to make this kind of distinction.

- Because proper names are persistent and connotations are ignored, if a term is a proper name and if in resolving it we discover other proper names for its referent, then all of these proper names are in a sense equivalent. Because we are using proper names only for finding and accessing their referents it does not matter which of the equivalent names we use. We call this property *persistent referential transparencey.* It has also been called *interchangeability* [Car56]. Naming schemes involving arbitrary indirection in which intermediate contexts can be arbitrarily changed do not satisfy our model of proper names because they do not have the interchangeability property.

- Strawson [Str59] made a comment that we do not require users of natural language to necessarily be able to resolve all of the names they can use, nor do we require them to remember why a name refers to a particular object. In contrast to this we do require that processors in our system be competent about what they know. Given a proper name a processor should be able to efficiently resolve it, either finding the referent or discovering that it no longer exists or is inaccessible. Given the representation of an object and a proper name a processor should be able to efficiently decide whether the name refers to the object. It may need the help of other processors to decide this but it will not have to perform any computations that depend upon the global state of the system. A processor possessing two proper names should similarly be able to decide efficiently whether they refer to the same object or not.

These properties constitute an operational criterion for deciding whether a particular naming scheme should be regarded as implementing proper names. While there may be many naming schemes that are consistent with these concepts, we adopt a version of what has come to be known as the *causal model of proper names* [Don72, Kri72]. The model is based upon the observation that while a proper name can be described in terms of the details of its implementation, its most important property is that there is a chain of causally connected events in the history of the system that link the event of using the name with the current state of its referent. It is this causal chain rather than the description of the implementation that is important.

In a similar vein, Morris [Mor73] argued that an abstract data type should not be thought of as a predicate that extensionally describes a set of data structures. If it were, then any data structure that satisfies that predicate may be thought of as an object of that type. Instead he argued that a data structure's history is more important than its structure. For the purposes of type checking, it is more important to know "... *where* it came from or *who* created it."

In the causally connected model of proper names a processor P can come to legitimately possess a proper name n of an object x in only three ways.

- Object x may have been located at P and P bound n to x.
- The name n was passed to P by another processor that legitimately possessed it.
- P created n as a synonym for another name m which it legitimately possessed.

Of these three ways of possessing a name we will only use the first two. While nicknames and aliases appear in some systems, our requirement that processors be able to efficiently compare proper names greatly diminishes their usefulness here.

To ensure that a processor that possesses a proper name will be considered competent it must maintain a data structure called a *context* to help resolve it. A context may do its job in one of several ways.

- The context at a processor at which an object is located must contain an ostentive reference for it in the form of a handle.
- The context at processor P can identify the processor from which it obtained the name. If this is allowed then that processor must also maintain a context for resolving the name.
- The context can identify a processor that is guaranteed to have a context containing "better" information than P about the object's current location. For instance, when an object moves, the old location can remain competent by identifying the location to which it moved.

The events of baptising objects, moving them, and passing proper names between processors are all causally connected to one another. A processor cannot pass a proper name unless it has received it in some previous event. Similarly, a properly named object

cannot be moved from a processor unless there was a prior event at that processor in which the object was baptised or in which the object was moved to it. Contexts with the above properties are witnesses to the existence of these causally connected chains of events. They provide at least one path from every processor that possesses a proper name to the processor at which the referent is currently located.

We chose the causal model of proper names because it is particularly well suited to our model of very large decentralized systems. It ensures that no processor can try to use a proper name to refer to any object that it could not communicate with, either because they are in different connected components of the network, or because of physical and temporal separation.

Since no two objects ever have the same representation at the same time their handles at that time must be different. A data structure containing a processor identifier, a handle, and a timestamp is therefore a viable candidate for a proper name. To decide whether two identifiers created in this way refer to the same object might require additional information. To make the comparison of proper names local and efficient we define a particular proper name called the *Canonical Identifier*, or CID, of an object to be the temporally first such proper name used to refer to it. We require that all proper names and contexts for an object have copies of its CID encoded within them. Furthermore we require that the object's representation be accompanied by a copy of the CID as it moves. We refer to the event of assigning a CID as an object's *baptism*.

The definition of CID's is for the sake of efficiency and simplicity in our exposition. Other schemes for implementing proper names [CM84, Sol85] could be used as well, but because these schemes allow the creation of nicknames and aliases the problem of deciding whether a name refers to a particular object or deciding whether two names are equivalent would be more difficult.

Because we are using a causally connected model of proper names, any consistent scheme for implementing them must be capable of sending a CID or a forwarding address when a proper name is passed between processors. Any such naming scheme could therefore use forwarding addresses for object finding.

1.6 Organization of the Dissertation.

This dissertation has three major parts.

The first part introduces objects, proper names, and the object finding problem. We review mechanisms that have been used for name resolution. We propose the use of forwarding addresses as a natural and efficient mechanism for object finding within the framework of our model.

The second major part is a series of analyses of the cost of using forwarding address protocols. We assume that in a real system most applications will be naturally structured so that the mechanism is used in an efficient way. This remains an assumption and for some applications may prove to be incorrect. To investigate the implications of this we perform average and worst case cost analyses for several object finding protocols using forwarding addresses.

The third part is a presentation of model implementations of forwarding address protocols. These model implementations are presented in order to make specific the relationship between naming and the definition of objects. In addition, we explore methods for extending the basic forwarding address mechanism to improve storage management and provide fault tolerance through replication.

The original contributions of this dissertation are the development of the model of objects and naming to make explicit the object finding problem and expose it to examination, the proposal that forwarding addresses are the natural and perhaps only viable general mechanism for object finding under such a model, the analyses of the costs of using forwarding addresses, and finally the extensions of the basic forwarding address mechanism to deal with issues such as storage management, administration, reconfiguration, and reliability.

1.7 A Chapter by Chapter Synopsis.

Chapter 1. This introductory chapter.

Chapter 2. We review the known methods for name resolution in general. We advance the idea that forwarding addresses are a good mechanism to use for object finding

as we have defined it. The disciplined use of forwarding addresses can simulate directories and lending libraries. In addition, the mechanism can be used in other, more general, ways.

Chapter 3. The disciplined use of forwarding addresses is no more expensive than any other method of finding objects. On the other hand, the undisciplined use of forwarding addresses can make some operations relatively expensive. In this chapter we analyze the costs of using forwarding addresses protocols, both in average cases and ammortizing the worst case costs over the lifetime of an object. One of the simple protocols that we examine generates random trees of forwarding addresses and our average case analysis technique improves upon the published estimate for the average depth of nodes in random labeled trees. The worst case analysis of another technique tightens the best published analysis of a sub-optimal algorithm for the set union problem [TvL84].

Chapter 4. The costs cannot be dealt with entirely by analysis. This chapter supplements the previous one by presenting some numerical and simulation results.

Chapter 5. We add detail to the model of Chapter 1 in preparation for presenting model implementations.

Chapter 6. We present a model implementation of a basic forwarding address mechanism for finding atomic objects in a system in which processors have stable transactional storage that is not destroyed by faults. Of special interest is the relationship between the definition of the identity of an object and the object naming and finding sub-system.

Chapter 7. With the basic mechanism it is difficult to reclaim the storage occupied by a forwarding address if the object it points to still exists. We explore extensions to the basic mechanism that enhance our ability to reclaim storage. One such extension, the use of *inexact reference sets*, explicitly represents the obligations of a processor to store things for other processors. It is particularly attractive because it is efficient and it gives a processor the ability to actively administer its resources.

Chapter 8. Distribution and replication are important for reasons of performance and reliablity. We discuss the extension of the previously presented mechanisms to handle reliably and efficiently distributed objects.

Chapter 9. We recapitulate our results and discuss open problems and possible future work in the area.

Chapter 2

Mechanisms for Object Finding.

In this chapter we review mechanisms for name resolution, both in the general case and when the problem is restricted to use only causally connected proper names. One of the techniques, the use of forwarding addresses, is particularly well matched to our model of objects and proper names. We define a generalized forwarding address mechanism and introduce several policies with which it can be used.

2.1 Basic Mechanisms for Name Resolution.

We begin by presenting the mechanisms by themselves. Real systems use name resolution and object finding strategies composed of a combination of mechanisms. We will review some of those combinations in the next section.

2.1.1 Addresses Encoded in Names.

If an object never moves and if causally connected proper names are used then the location of the object can be encoded in the name or sent along with it. Moveable resources in the system are implemented at more abstract levels of naming and resource definition.

2.1.2 Searching the Network for the Object.

A processor with a name or description can attempt to use that term by searching the network for an object to which it refers. The search can be done by polling the other

processors, by broadcasting, or by a combination of these methods.

In a very large system the cost of a global broadcast is extremely high. On the other hand, local area networks such as Ethernet have been designed to make local broadcast relatively efficient. A technique known as *directed broadcast* [Bog82] allows a searcher to search the global network by polling each local area net in one message with a broadcast.

In the worst case, searching used alone can cost up to $N - 1$ messages where N is the number of processors in the system. The average cost depends upon the kinds of locality assumptions that are made.

2.1.3 Posting the Name and Location of the Object.

An object can post its name or a concise description of itself together with its current location to all the processors that may want to access it. The latter store these pairs in an associative table for later use. If any of these processors wish to access the object then its location can be discovered by looking it up using this context. If this posting is a necessary part of the act of creating or moving an object then by our definition of "object" these table entries must be regarded as part of the objects's representation. If the table entries are regarded as merely hints and can therefore be done after the object's location changes then this is not the case.

By itself posting requires at least $N - 1$ messages to inform all of the processors in the system of the object's location.

2.1.4 Establishing a Rendezvous.

Searching and posting can be combined. The processor at which the object is located can post its name and location to a number of intermediary processors willing to respond to attempts to access it. The searcher and the object establish a *rendezvous* when the searcher interrogates a processor to which the the object's name has been posted.

The cost of establishing a rendezvous depends upon how much information each processor has about the rest of the system:

- Whether the participating processors have established *a priori* conventions to select specific rendezvous locations or the order in which they will be selected.

- Whether or not they have knowledge of the connectivity of the network and are allowed to form conventions for the use of this knowledge.
- Whether or not they are allowed to perform computations on names and processor identifiers to decide where to rendezvous.

In the case in which processors are allowed none of these the only strategy available is to choose intermediaries at random. The object's name and location are first posted to p different processors. The searcher then polls randomly chosen processors until a positive response is received. Because an efficient searcher never polls another processor twice the problem of establishing the rendezvous is one of sampling without replacement. The probability of having established a rendezvous is given by a hypergeometric probability distribution. If we let s be the number of processors polled, and $\mathcal{N}$ be the total number of processors and if we assume that p is a small fraction of $\mathcal{N}$ and that $\lambda = \frac{ps}{\mathcal{N}}$ is "of moderate magnitude" then the probability that a rendezvous has *not* been established is approximated by the Poisson distribution $p(0;\lambda) = e^{-\lambda}$ [Fel68, p. 172]. This probability is reduced to less than $1/e^{\epsilon}$ for any constant ϵ when $ps > \epsilon\mathcal{N}$.

If processors are allowed to establish conventions that use computations on processor identifiers then they can establish the rendezvous in a constant number of messages. For instance each can simply send a message to the processor in the system with the lexicographically least identifier. In effect, they turn it into a "rendezvous server". Although each rendezvous costs a constant number of end-to-end messages, a difficulty with this strategy is that there will be very high traffic at the "well known" processors that serve as rendezvous. In a very large, highly concurrent system the rendezvous will have to occur at a large number of processors in order to avoid bottlenecks.

The burden of serving as a potential rendezvous point can be spread out by prearranging to choose rendezvous processors based on a function of the name of the object. Mullender and Vitáni [MV85] explore the complexity of this problem for a variety of strategies and constraints upon the storage each processor is willing to commit to the problem.

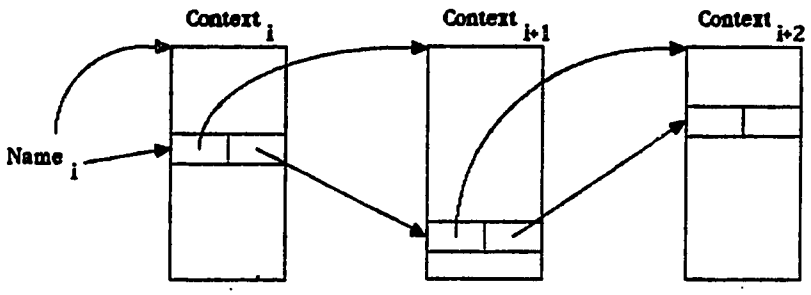

Figure 2.1: The resolution of name and a context yields another name and context. Resolution can be iterated until a primitive, or ostentive, reference is found.

One problem with using rendezvous as an object finding strategy is that it uses resources at the rendezvous processors. This requires the cooperation of the administrator of that processor. If the rendezvous processor is not an interested participant in the application then some form of compensation for that cooperation is appropriate.

2.1.5 Chained Contexts with Known Addresses.

One way of viewing name resolution is that it consists of a sequence of stages, each of which is the translation (mapping) of a symbol and a context in which it is used into yet another symbol and a context in which the new symbol can be resolved [Wat81] (Figure 2.1). To access the referent there may be a sequence of such resolution stages ending in one whose result is some kind of primitive symbol for the object. Comer and Peterson [CP85] present a formal model for describing naming systems based upon this resolution strategy.

If the contexts are represented as concrete data structures that never move then their names can contain their locations. Each stage of name resolution can proceed by going directly to the location of the context for that stage. The contexts form a chain of indirection to the object.

Arbitrary directory, or index, structures can be constructed using this form of indirection to chain multiple contexts.

2.1.6 Caching Recent Locations.

One thing that can always be done is for a processor to cache information about objects that it has accessed recently. By caching locations of objects and contexts one can avoid repeating an expensive search.

The locations of the rendezvous processors for each object can be cached as can the object's last known location.

2.1.7 Forwarding Addresses

A particular case of using chained contexts for finding moveable objects is the use of chains of *forwarding addresses* for each object. This approach has been used to find migrating processes in DEMOS/MP [PM83] and Halstead [Hal79] proposed using them in a lattice-connected distributed system. A forwarding address held by one processor that points to another may be thought of as a directed edge between them, and the collection of forwarding address chains can be viewed as a distributed directed graph.

In Halstead's work each processor maintains a single forwarding address for each object it can reference so the forwarding address graph for an object is a tree. All communication related to that object is routed over the bidirectional edges of that tree. All routing information is distributed over the tree and a processor accessing a remote object does in general not learn its location. No attempt is made to modify the tree to improve the efficiency of access. Thus, even if the the object is located at a processor adjacent in the lattice to the processor seeking it, messages between the two may still take a circuitous route determined by the pattern in which the the object has moved and the pattern of name distribution.

In DEMOS/MP when a process is migrated from one processor to another it leaves behind a forwarding address. When the migrated process is accessed the originator of the operation learns of the new location and updates its information.

Forwarding address mechanisms can be distinguished from cached information in that the latter is regarded as a "hint". The cached information can be invalid and the system can be forced to revert to another strategy for finding the object. On the other

hand, we regard a forwarding address to be a form of proper name that identifies the location of the context in which it is to be resolved. In particular, the processors to which forwarding addresses point and at which they can be resolved are constrained to be past locations of the object they name.

2.2 Name Resolution in Real and Proposed Systems

Real systems use a combination of these basic techniques. We review a few here and point out how the techniques are combined.

2.2.1 Distributed File Services.

In some models of distributed file services the location of an object is not a processor but rather a storage module such as a disk pack. In the Cambridge File Server [Dio80, MD81] each file is represented as a tree of disk blocks with the leaves containing data and the non-leaf nodes serving as indices to their children. The unique identifiers for files stored in this system identify not only the disk upon which the file is stored but also the physical location on the disk of the root of the tree representing the file. A file can be moved from processor to processor only by physically moving the disk. The file cannot even be relocated on the same disk.

In the Xerox Distributed File Server [SMI80] the unique identifier used as the proper name of a file contains the manufacturer and serial number of the disk pack upon which the file resides. Again, objects (files) are moved only by physically moving the disk pack on which they reside. The file servers keep an index of the the names of the currently mounted disk packs for all the file servers currently part of the file storage service. Thus, once an application has found a server by performing a search, that server has a context that enables it to redirect a request to the correct server.

Closely associated with each file server is a directory server that implements the notion of user directories that contain user sensible, descriptive string names for the files that have been cataloged in the directory. The higher level abstraction created by the directory allows one to define entities that can move around.

In these systems a combination of searching and posting is used by a processor to learn the addresses of the file and directory servers.

In the Apollo Domain (TM) system [LSHL82] objects (files) are internally named with unique identifiers (UIDs) that contain the name of the processor at which they were created. On top of this is a network–wide hierarchical directory system that maps string names into UIDs. At the level of the programmer interface an object can be "moved" from one processor to another, but this is implemented by creating a copy with a different UID at the destination processor and modifying the directory to reflect this change. As in the other file services we described, it appears that the only way for an object to move from one processor to another and retain its UID is for the physical medium on which it is located to be moved. This is a very rare occasion.

Because object motion is rare and because looking up an object in a local table usually incurs the cost of one or more disk accesses, the method used for finding an object given its UID is to first interrogate the processor named in the UID. If this processor no longer has the object but knows where it is then responds to the request with a forwarding address pointing to the current location. Otherwise, the access attempt fails and the processor originating the operation may consult a local "hints" database to find other processors to interrogate.

2.2.2 Generalized Name Services.

Our model of causally connected proper names requires that the passing of proper names between processors be internally witnessed by the system. While a human user of this system could refer to an entity by picking its proper name out of a menu, a text string that a user types on keyboard is not regarded as a proper name within the narrow context of the model. The model does not differentiate between randomly (or nondeterministically) generated text strings and those that are passed outside the system or are otherwise not witnessed. For example, a user at one location might telephone another user to inform the latter of the name of some entity, perhaps the login name of some account and its corresponding password. The system cannot differentiate between this case and the case

in which a system cracker guesses the same information.

Text string names are, however, important to distributed computation. Databases called indices, catalogs, or directories that map text strings into addresses or other forms of proper name will continue to be needed. As such databases have grown, incorporated authentication checking, and become distributed they have come to be known as *name services*. To paraphrase Lampson [Lam85], a name service is a name resolution database optimized to provide very reliable and efficient accessibility to objects that move about as often as an electronic mailbox of a user of a computer system.

Grapevine [BLNS82, SBN84] is a message delivery service that contains a name service for its users. Names are of the form "user.registry". Each registry is a replicated database that serves as a context for resolving the name "user". When the address of a named object is changed the changes are posted to all Grapevine servers that store copies of the registry encoded in the name. Furthermore, all Grapevine servers store a copy of a "root" registry to which the adddresses of all Grapevine servers are posted.

A processor attempting to send a message may have cached the address of the addressee. If so then the first thing tried is to send a message there. If that fails then the address needs to be gotten from the Grapevine service.

A Grapevine server can be found either by using a cached address or by searching. Once one server is found the name is resolved by following the posted contexts. First a server that stores a copy of the approptiate registry is found and then the object itself is found from its posted address.

The Clearinghouse [OD83] is another distributed name service that uses the mechanism of partitioning the name space among various authorities to achieve a degree of decentralization.

Terry, in his dissertation [Ter85], discusses the performance of these name services with an emphasis on the effects of caching and of locality on performance.

2.2.3 Eden.

Eden [LLA*81, ABLN83] is a distributed object–oriented system in which objects known as *ejects* are named with capabilities that contain a unique identifier for the eject as well as a vector of "rights". In general, each eject is composed of two parts. There is an active part that can handle requests to access it, and there is a passive part consisting of a passive representation of the object as of the last time it was checkpointed. An object that has never been checkpointed will not have a passive part. An object can passivate itself by deleting its active part. If a request to access a passivated object is made an active part is reconstructed from the checkpointed representation.

To find an eject a processor p first looks in a cache to determine where the active part of the eject was last seen by p. If this fails then it looks in the eject's capability for an Eden host identifier. This is used as a hint for finding the host (called a checksite) at which the passive part is stored. If this fails then the seeker sends a multi-cast to all checksites. Finally, if no reply is received the seeker broadcasts the request to all other processors. This last resort is included to handle the case in which the seeker has been given a capability for a newly created object that has not created a passive representation by checkpointing itself [San85].

The causal connection between Eden capabilities and their referents is used only as a hint for finding the eject's checksite.

2.2.4 Gifford's Decentralized Storage System.

Gifford [Gif82] proposed a paper design for a decentralized storage system. A structure called an *index* provides a level of indirection by mapping from string names to internal identifiers called *references*. Indices are objects similar to Unix directories and can be used to build similar hierarchies. Unlike Unix directories they can be implemented as replicated databases.

The root of the hierarchy is not at any fixed location and can be found only by searching for it. Once the root is found all other objects in the system can be found by following an appropriate chain of indices.

2.3 Methods for Object Finding with Proper Names.

We turn now to the problem of object finding in a very large and dynamic system. Our assumptions about the extreme size of the system and the dynamic nature of the population of objects make some of the above mentioned techniques unsuitable for object finding:

- Searching a very large network for a large number of small, mobile objects is far too expensive. Similarly, the cost of posting the name of a small, mobile object to any significant number of processors far outweighs the cost of moving and accessing it.
- Applications must be able to proceed autonomously. The mechanism for object finding should therefore be fully decentralized. If a small object can be sent from one processor to another in a single message then it shouldn't be necessary to automatically inform a third of the object's existence and motion. In this case requiring the use of intermediaries as rendezvous points is inefficient, and increases the application's vulnerability to faults. Another problem with using system-assigned rendezvous processors is that it is necessary to pay for the service they render and they can become bottlenecks.
- Our assumption that objects will move precludes the use of fixed addresses encoded in their names.

On the other hand, the causally connected model of objects and proper names ensures us that in the absence of faults there will always be a chain of events connecting any instance of a proper name and the current state of its referent. By requiring that processors be competent about what they know about proper names we ensure that these chains of events are represented as chains of contexts that can be used to find the referent. The existence of these chained contexts means that it is not necessary to use arbitrary searching and posting. Furthermore, the processors in these causal chains are by definition participants in the application and are therefore in a sense "interested" in the object. Because they are willing participants in the application there should be less of a problem with paying for their cooperation in helping to find the object. The problem of processors that "lose interest" is, however, important and we will discuss it in Chapter 7.

It is necessary to be able to manage these chains of contexts so as to make object finding efficient and to avoid bottlenecks. Our insistence on persistent referential transparency means that if a processor P possessing a proper name X for some object discovers another proper name Y for the same object then P can correctly use either X or Y. P is free to make this choice based upon an estimate of the reliability or cost of resolving each of the names. We can make the comparison of proper names both efficient and local by requiring that all proper names for an object contain a canonical identifier (CID) for that object.

2.3.1 A Generalized Forwarding Address Mechanism.

To focus on the problem of efficiently maintaining causal chains we will concentrate on the use of forwarding addresses to name objects. As described above a forwarding address is a proper name that contains the name of a processor that has a context within which that name can be resolved. That context ie either another forwarding address of a handle for the referent. Unlike more general schemes of chained contexts, forwarding addresses are constrained to point only to processors at which the representation of the named object has been located. Usually each forwarding address either points to the processor at which the named object is now located or it points to a processor that has a more recent forwarding address for it. The exception occurs when the object no longer exists. In this case a forwarding address can point to a processor that has no information about the object. From this it can be inferred that the object has been deleted so we can interpret the absence of a forwarding address as being a special **NULL** forwarding address that signifies that the object no longer exists.

Our generalized forwarding address contains the following information.

- It must contain a copy of the CID of the named object. This is so that a processor can decide whether two forwarding addresses refer to the same object or not.
- It contains a non–empty ordered set of processor identifiers L. In the simplest case for atomic objects L will name exactly one processor. The location of a distributed object is, by definition, the set of processors that simultaneously hold handles for it. In this case the contents of L names a subset of the location.

- Given two forwarding addresses for a single object a processor must be able to distinguish which of them is more recent. To do this a forwarding address contains a timestamp T. The timestamps for each object must increase monotonically, but they need not be correlated with the timestamps of other objects.
- A forwarding address may contain the identifiers of other processors at which the object has been located and which continue to bear a special relationship to it. For example, it might name the creator of the object, a processor that the object expects to visit often, or a processor that the object expects to communicate with regularly.
- A forwarding address may contain additional information to facilitate efficient storage management.

The interpretation of a forwarding address is: "At time T the set of processors L had handles for the object named by this CID. Furthermore, if that object still exists then the members of L will have a forwarding addresses for it." We require that each processor that has a proper name for an object also have forwarding address for it. This can be enforced by sending a forwarding address along with every proper name transmitted between processors. We also require that each processor p at which the object has been located keep a forwarding address for the object for as long as there is a possibility that some other processor may have a forwarding address that points to p.

In most cases each processor will keep a single forwarding address for each object it can name. In addition they may choose to keep others in reserve for the purposes of fault tolerance.

When an object moves a new forwarding address pointing to the new location is created with an updated timestamp and copies of this forwarding address are kept by all of the processors involved in the move, replacing their old forwarding addresses for that object. A processor at which an object is located is thus guaranteed to have a copy of that object's most recent forwarding address.

A new forwarding address need only be created when there is an attempt to move an object. For that reason it is sufficient to use a count of the number of times an object has attempted to move as the source of timestamps for that object. Timestamps that more closely mimic physical time may, however, be desirable for purposes of identifying

old and probably obsolete information.

From time to time a processor will receive new forwarding addresses for objects it can name. It may choose to keep new ones and/or delete an old one as a result. We require that no forwarding address be deleted by a processor unless it is replaced by a more recent one for that object. Because the information at each processor can only become more current, the most recent forwarding address at processor P for an object that has been located at P is guaranteed to be at least as current as the most recently created forwarding address for that object that points to P. This implies that a processor pointed to by a forwarding address F must have a forwarding address F' at least as recent as F. The only case in which $F' = F$ is that in which the processor named is still the location of the object. Thus, the timestamps along any chain of "most recent" forwarding addresses increase monotonically until reaching the processor(s) at which the object is located.

We can view the structure of the forwarding addresses for an object X as a directed graph $G = (P, E)$ where the vertices P are processors and ordered pair of processors (p, q) is in the set of edges E if and only if processor p has a forwarding address that points to q. Processors that currently store the representation of X have self-loops that point to themselves.

Since the timestamps along any chain of forwarding addresses increase monotonically until the current location of X is reached, G must consist of a single connected component containing all processors that can refer to X, and a set of singletons that cannot. Furthermore, the monotonicity property ensures that all cycles in the graph are contained entirely within the set of processors at which X is currently located.

In the simple case in which each processor keeps at most one forwarding address per object and in which all forwarding addresses name exactly one processor, the graph of forwarding addresses for each object is a tree rooted at the current location of that object. We call this a *forwarding address tree.*

The forwarding addresses we have defined are a decentralized mechanism that can be used to ensure that all processors that can use a proper name to refer to an object will be able to find it. Each of these processors will have a forwarding address and in

the absence of failures there will be at least one path of forwarding addresses from the object to the current location of the object.

2.4 Policies for Using Forwarding Addresses.

The forwarding address mechanism is decentralized and, given the rules for creating new forwarding addresses when an object moves and for replacing old forwarding addresses only with newer ones, it correctly ensures that all processors with proper names for an object will be able to find it. While correctness depends upon the basic mechanism, performance will depend upon the policies with which it is used. For instance, the naive use of forwarding addresses may cause some operations to be expensive. For example, an object could visit N different processors in turn to create a path of forwarding addresses of length $N-1$. There can therefore be an object finding operation that requires that $N-1$ messages be transmitted. Because of this it is important that policies for the efficient use of forwarding addresses be developed. This can be done by preventing such expensive operations from occuring or by ensuring that they occur rarely and that the ammortized or average costs of using the mechanism are kept low.

These policies can be implemented in two places: in protocols defined by the system and thus applicable to all applications, or in the applications themselves. Some of these policies we discuss introduce a form of centralization. In these, some processors will take on service roles with respect to the object and will be required to use an abnormally large part of their resources to help others to find it. There is a potential that these processors could become bottlenecks. The decision whether or not to use these strategies should therefore be left to the applications themselves. Other policies for the efficient use of forwarding addresses occur as a natural part of the application and require no special effort to implement.

2.4.1 Forwarding Address Distribution

In the basic mechanism introduced above forwarding addresses are transmitted between processors whenever proper names are. This applies both to the case in which they are

sent by themselves and when they are encoded in the state of an object that moves. In what other circumstances should forwarding addresses be sent?

Path Compression

One possibility is that when a processor replaces a forwarding address F with a newer one F' that a copy of F' be sent to the processor(s) pointed to by F. If this rule is applied by each processor along a path of forwarding addresses then the effect will be to replace all forwarding addresses in the path that are older than F'. This induces a form of path compression as illustrated in Figure 2.2. In Chapter 3 we analyze the effects of adding path compression on the cost of using forwarding addresses. Because this mechanism does not assign a special role to any processor this is a policy that can be implemented by the system for all applications.

Keeping Certain Processors Well-Informed.

Applications may want to make sure that the forwarding addresses held by certain processors are current. For instance, an object X might want to make sure that the processor that holds object Y will be guaranteed that it can find X by following a path of at most two forwarding addresses. To do this X can send Y a message containing its name immediately after each move. This will cause an up–to–date forwarding address to be sent from X's location to Y's location. Usually Y's location will know X's current location. When the move and the attempt to access are so close together in time that the information about the move has not yet propagated back to Y's location the access may have to follow a path of length 2.

We generalize this strategy to guaranteeing that a processor p need only follow a path of length k where $k \geq 2$. To ensure this we need only to have the object report its location to p after every $k-1$ moves.

This strategy might be used in the object–oriented system of the Humongous Corporation. An intelligent document may report back to its originator each time it takes a significant step as it makes its rounds of the automated bureaucracy. The originator

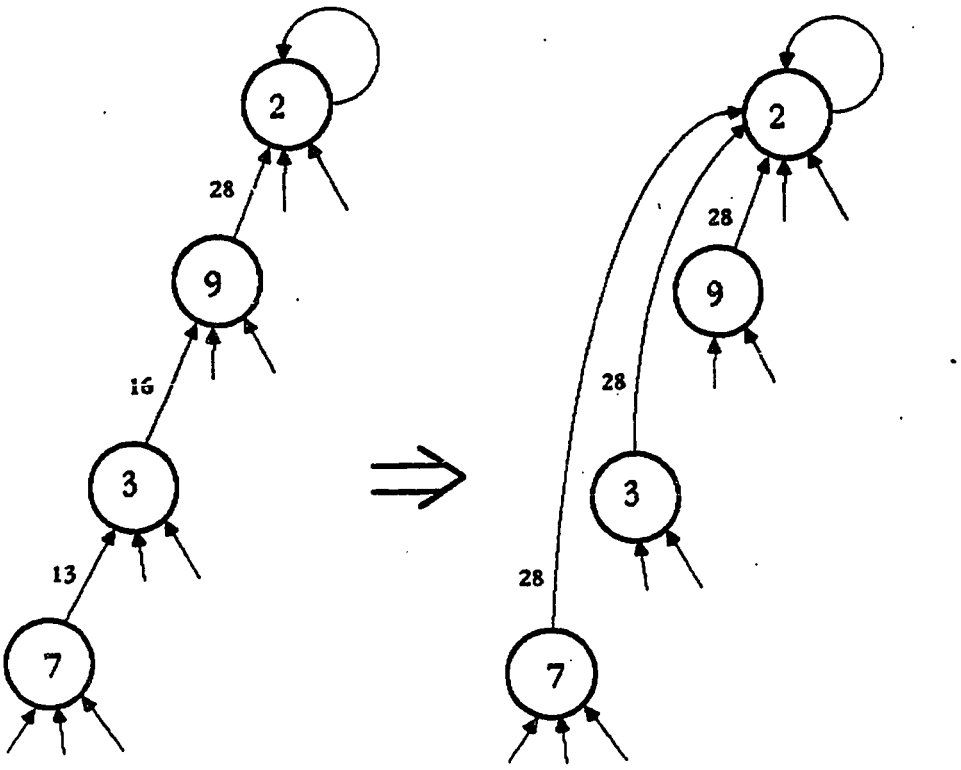

Figure 2.2: Processor 7 receives a forwarding address pointing to processor 2 and containing timestamp 28. (Edges representing forwarding addresses are labeled with the appropriate timestamp.) By passing the new forwarding address along the original path to 2, the path to the object is compressed.

thus knows where it is and can access it efficiently in order to expedite its progress or to monitor the operations that have been taken on it. In Figure 2.3 the originator is appraised of a travel requisition's progress.

Another way of ensuring that a particular processor knows the location of object X is for the processors that access X to designate one of themselves to be a repository of a recent forwarding address for X. Whenever a processor discovers that X has moved it informs the repository by sending it a copy of the new forwarding address. Note that the repository is not guaranteed to know the "truth" about the X's location, rather it is guaranteed only to have recent information.

Using Knowledge of Well–Informed Processors.

If an application chooses to ensure that a particular processor H will be able to find object X with a constant number of messages then this fact can be used by all processors participating in the application. Soon after each time X moves H is sent the latest forwarding address for it. All processors can first attempt to access X at the processor named in the forwarding address they have stored. If X has moved they can then thus get a newer forwarding address from H. With this strategy an application can guarantee that all processors can find X using a constant number of messages. This protocol is illustrated in Figure 2.4. In this example, each processor keeps an old forwarding address that points to processor H, in this case, the processor at which X was created. In addition, each keeps a copy of the most up–to–date forwarding address it has seen. These are represented by the heavier edges. After the object moves, a message is sent to H informing it that the move has occured. The figure illustrates the worst case situation in which the object has just moved to processor p and the path to p from q through H is of length 3. The total number of edges used is 4.

This is analagous to what happens when people move. Relatives, close friends, financial institutions, and employers are typically informed when one changes one's address. Other acquaintances and institutions may have to rely on forwarding addresses or on knowing the identity and location of some one, such as a parent or employer, that is

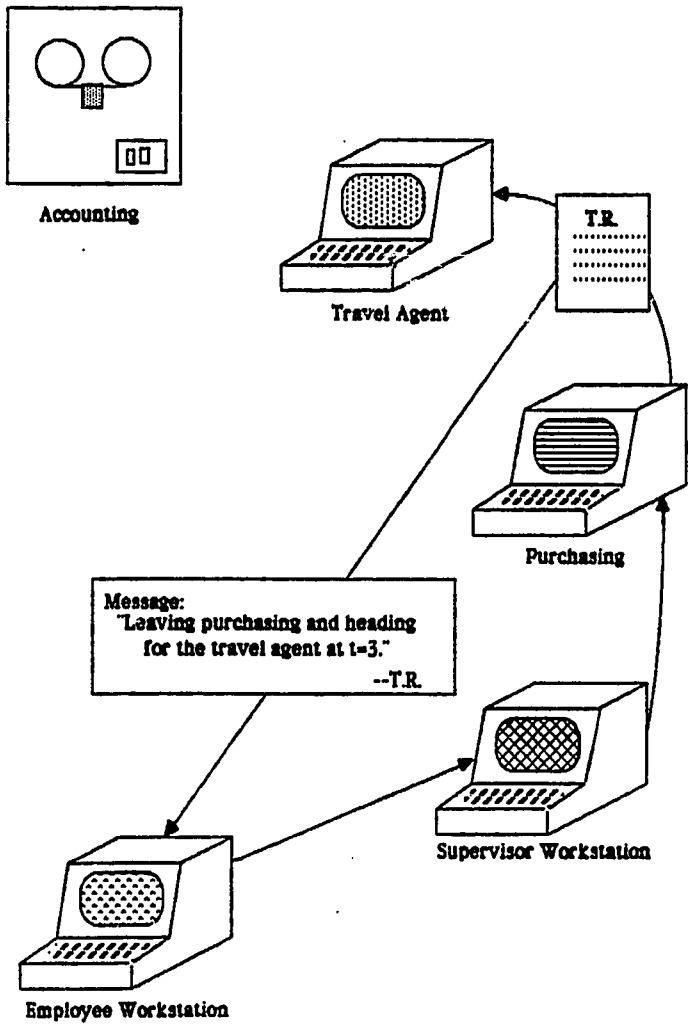

Figure 2.3: An intelligent travel requisition form reports its bureaucratic progress to its originator.

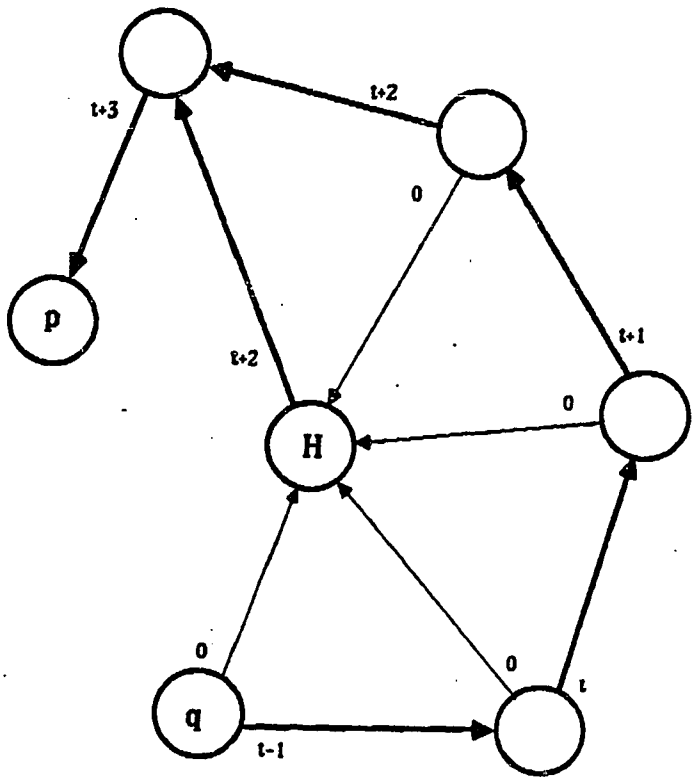

Figure 2.4: If H is kept informed of the object's location then in the worst case processor q can find it with 4 messages.

likely to have the correct address.

This strategy uses the stored forwarding address as a form of cached location. It is more than a hint because if H has failed then X can still be found by following the path of the most up–to–date forwarding addresses. This works as long as X is not at H and there is a forwarding address path in which H does not appear. The reliability properties of forwarding address mechanisms will be discussed in more detail in Chapter 8.

2.4.2 Decentralized Objects.

Some decentralized objects may have parts of themselves that move slowly if at all. For example, in Eden an eject may be activated at many different hosts during its lifetime. The checksite of that eject, however, almost never moves. It can be much cheaper to find an object by finding its slowly moving parts than to try to find the rapidly moving parts directly. The extreme example of this strategy is to place one or more handles for an object in fixed locations.

Recall that all data structures that can affect or can be affected by a change of an object's abstract state are defined to be part of the object's representation. An object's handle is part of its representation because it encodes the truth about its current location. These properties of our model of objects distinguish the case in which a handle is known to be in a fixed location from that in which the location of a recent forwarding address is known. In the latter case the object can move without sending any messages to the location of the forwarding address. The transmission of the new forwarding address after the move has completed is an independent operation. In the case of the handle at a fixed location the handle *must* be updated as part of the move.

Similarly, if a directory or catalog data structure contains entries which, like handles, partially *define* an object's location then they must be regarded as part of the object's representation. This distinguishes processors that are known to have recent forwarding addresses from other forms of servers whose states actually defines the location of the objects they help find.

2.4.3 Object Motion.

The pattern in which an object moves also affects the cost of using forwarding addresses. Efficient patterns may occur naturally as an inherent locality property of the application or they may be used deliberately in order to ensure that it is cheap to find objects. As in the case of deciding to keep certain processors informed of an object's location these patterns can introduce a form of centralization within the application and should therefore be chosen by the application rather than being imposed by the system.

Homes, Offices, and Libraries.

In many applications it will be natural for an object to visit particular processors on a regular basis. For instance, in our example the Humongous Corporation's automated factories have depots known as *tool cribs* that act like lending libraries for tools. A typical pattern of motion for a tool will be from the tool crib to a manufacturing robot, back to the tool crib, to another robot, back to the tool crib, to the toolmakers' shop for maintenance, back to the tool crib, etc. The maintenance record for that tool will accompany it from processor to processor around the factory, always returning to the processor that keeps the tool crib inventory.

Other examples of this pattern of object motion can be found in database applications in which centralized repositories hold collections of objects that can be checked out of the database by the workstations of persons wishing to work on those objects. Engineering databases for complex projects such as VLSI chip design [Kat82] or software development [Sch82] are prime examples of this mode of operation. A chip design or a software release is represented by a collection of objects, each of which is a version of a component module. Version objects are checked out of the database, manipulated at engineering or programming work stations, and are then checked back into the repository.

These patterns of motion have an effect similar to some of the forwarding address distribution strategies. They ensure that there are one or more processors that are guaranteed to be able to find the object using a chain of forwarding addresses of constant

length. Knowledge of this fact can then be used by other processors to ensure that they in turn can access it with constant cost. Even if other processors do not explicitly use this knowledge, the fact that the object will be "seen" often at the repository can keep the average length of a chain of forwarding addresses short.

2.4.4 Adding Maintenance Operations.

Another strategy that can be used to ensure that forwarding address paths used by attempts to access an object will be short is to add additional operations for maintenance purposes. Such operations can be performed during periods of low usage. One reason for using timestamps that approximate physical time is that they make it easy to identify forwarding addresses that haven't been updated recently. Processors can attempt to follow their paths specifically to discover where the referent currently is rather than to access it. This can also be done to discover which forwarding addresses are completely useless because their referents no longer exist. Such forwarding addresses can themselves be discarded.

The use of path compressing maintenance operations can have the effect of increasing the total amount of work done over an object's lifetime in exchange for decreasing the amount done during periods of high demand.

Path compression can be delayed by giving the messages containing forwarding addresses for path compression a low priority. The possible effects of this upon system performance is discussed in Chapter 3.

2.4.5 Arbitrary Use of Forwarding Addresses.

The policies that we have been discussing are available to an application that uses the generalized forwarding address mechanism to find its objects. Some of these policies introduce forms of centralization with respect to the application. Choosing which processors will play a special role in this centralization should be decided autonomously by the individual application and not be imposed by the system.

If these policies are made available then each application can pick and choose among

them to put together an appropriate policy for finding its objects. Some applications might choose to use only the basic mechanism without using additional performance enhancing policies. It is interesting to analyze this situation because it gives us a basis against which we can evaluate the efficacy of using enhanced protocols. We analyze the arbitrary use of forwarding addresses in Chapter 3

2.5 Summary.

Of the techniques that have been used for name resolution, the use of chained contexts in fixed and known locations is the one most suited to object finding using proper names in our physical analogue model of objects. In particular, we require that objects move "continuously" in a causally connected chain of events and proper names be propagated only through similar chains. These causal chains of object motion and name passing events can be witnessed by chains of forwarding addresses.

The use of forwarding addresses as an object finding mechanism is further encouraged by the fact that it is decentralized and localized to the processors participating in an application. The cost of using it is therefore independent of the total size of the system. This is especially important in our anticipated world of very many small and mobile objects.

There are various policies that an application can choose to use to enhance the performance of the basic mechanism. Because many of them introduce a form of *ad hoc* centralization to the object finding process the choice to use them should be made internally by the application itself rather than being imposed externally by the system. Since many applications exhibit an inherent centralization these policies may occur as a natural part of the application rather than as the result of a conscious decision.

One enhancement to the basic mechanism that does not introduce centralization is the use of an updating operation that compresses paths of forwarding addresses. It may be a reasonable strategy therefore to include path compression as an optional part of the basic mechanism provided by the system.

Chapter 3

The Cost of Using Forwarding Addresses.

We have argued thus far that forwarding addresses constitute a simple and natural decentralized mechanism for finding moveable objects using causally connected proper names. Whether a location service using a forwarding address protocol can really be practical, however, depends upon many issues, not the least of which is performance. In Chapter 2 we discussed several policies that introduce a form of *ad hoc* centralization and ensure that in the absence of faults an object can be found using a small constant number of messages. For a variety of reasons not all applications will choose to use these policies.

In this chapter and the one following we investigate the performance of simple forwarding address protocols used in an arbitrary way. In this one we define the protocols and derive some analytic bounds on the cost of using them. In Chapter 4 we present numerical evaluations of those bounds graphically and compare them with the results of simulations.

3.1 An Abstract Object Finding Problem.

To study the complexity of object finding using forwarding addresses we first define a more abstract and formal version of the problem. In the abstract problem there is a single atomic object R, a *resource*, that is shared among a subset of cardinality N of the

processors in the system. Each of the N processors can access R and R can move to each processor. Let $S = (V, E)$, an undirected graph, be our abstraction of that part of system that shares R. The vertices V of S represent the ***processors*** and are labeled with the integers from 1 to N. The edges $E \subseteq V \times V$ represent a network of bidirectional, logical ***communication channels*** connecting these processors.

As stated in Chapter 1 we assume that at the level of the logical network any pair of processors can directly exchange messages. S is the complete graph on N vertices. While the physical interconnection structures of large distributed systems are usually partially connected, the model presented to an application program is usually that of a completely connected logical network. Even though the actual cost of transmitting messages between pairs of processors may vary greatly depending upon the distances between them and the routes chosen, in our simplified model we will treat them as though all the costs are the same.

At any time R is located at exactly one of the processors. A $move(x)$ operation moves R from its current location to processor x. Initially R is located at processor 1 and all other processors have forwarding addresses for R that point there. An $access(y)$ operation causes processor y to attempt a remote access of R by dispatching a message into the network with the intention that the system deliver it to the processor at which R is located.

A particular abstract object finding problem is the abstraction of a distributed application program running on the set of processors S. It is a mechanism that generates sequences of move and access operations that the system is required to execute on-line. An ***instance*** of an object finding problem is a particular sequence generated by that mechanism. A ***object finding protocol*** is a specification of how move and access operations are to be performed in S, along with a specification of the initial states of the processors. The protocol is not allowed to change or augment the sequence of operations generated by the application. This prevents the protocol from introducing *ad hoc* centralization.

Our measure of the cost of an operation is the number of (logical) network edges over which messages are transmitted in performing that operation. Each of the protocols

we analyze transmits a small, fixed number of messages over each of these edges in each operation. Our cost measure therefore approximates the number of point-to-point messages transmitted without getting into the details of low level message transmission protocols. The cost of executing a sequence of operations is the sum of the costs of the individual operations as though they were executed sequentially.

The abstract problem uses the simplest form of the forwarding address mechanism. Each processor keeps a single timestamped forwarding address for R. Each timestamp is the number of times that R had moved when the forwarding address was created. A forwarding address containing timestamp t thus points to the processor to which R moved on its tth move. When R moves from processor x to processor y, at least two forwarding addresses must be changed: both x's and y's forwarding addresses must point to y. Whenever a processor is sent a forwarding address in a message it compares time stamps to decide whether or not to replace its old forwarding address with the newly arrived one. Even if it can perform this replacement, certain protocols may choose not to.

Using these rules the current location of R always has a forwarding address pointing to itself. By the replacement rule any other processor x has a single forwarding address pointing to a processor at which R was located after the last time it visited x (if ever). The time stamps along any directed path of forwarding addresses thus increase monotonically and there are therefore no cycles. Since the out degree of each processor is 1 the resulting graph is a rooted tree $T_t = (V, E_t)$ where $(x, y) \in E_t$ iff processor x's forwarding address for R points to y. The root is the current location of R. In the initial state of the abstract problem the object has been created at processor 1 and all N processors have been given the appropriate forwarding address. In the initial state the tree is therefore of height 1.

3.1.1 *Ad Hoc* Centralization.

We saw in Chapter 2 that there is nothing to prevent an application from using the forwarding address mechanism in a disciplined way to create a kind of *ad hoc* centralization

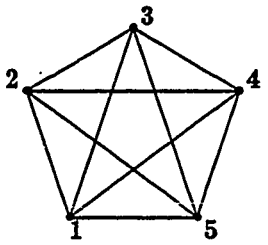

Figure 3.1: $S = K_5$, the complete graph on 5 vertices.

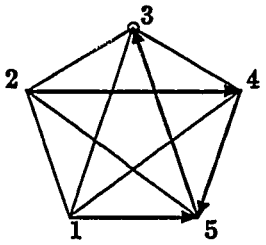

Figure 3.2: A forwarding address tree in S and rooted at 3.

that achieves a constant cost per operation. This can be done as long as it can find a processor willing to tolerate increased message traffic and possible congestion.

In order to study the decentralized use of forwarding addresses we prevent the system itself from assigning a distinguished role to a processor. In our abstract version of the problem this takes the form of a prohibition against implementations of the *move*(x) and *access*(x) operations that involve processors other than those on the path of forwarding addresses from x to the (old) location of R.

3.1.2 Three Forwarding Address Protocols.

Reasonable protocols for using forwarding addresses will allow each processor to freely update its forwarding address for R whenever it receives new information. In addition, they may or may not choose to induce path compression by having processors send new forwarding addresses to processors that were pointed at by old ones.

The updating of the forwarding address of a processor initiating an *access* occurs at the successful completion of that operation. It is possible therefore that there will be other operations that will not be able to take advantage of that update. The *access* might fail before the originator learns the location of R. Similarly, if several *accesses* are initiated concurrently then it may be that none of these operations can take advantage of changes to the forwarding address tree triggered by the others.

In addition to the circumstances that might prevent or delay the updating of the

forwarding address of a processor initiating an *access*, path compression may be delayed further by the slow propagation of maintenance messages. For instance, since path compression can double the actual number of messages sent compared to a simpler protocol it may be reasonable to send maintenance messages at a lower priority than others. The completion of path compression might therefore be delayed until a period of low system utilization. If this occurs, then a path compressing protocol would behave like a non–path compressing protocol during periods of high utilization.

Our analytic techniques are not able to deal with concurrency and delayed maintenance directly. In order to investigate the consequences of the concurrent execution and the delay of maintenance we instead analyze three basic protocols that incorporate varying amounts of maintenance on the forwarding address tree between operations. All three protocols we analyze use a simple *move*(x) operation that moves R from its current location, y, to processor x along an edge of S. Since only x and y know that the operation has been performed this changes exactly two of the edges of the tree: y now points to x, and x points to itself. Although two edges of the tree are changed, data is transferred over only one. The cost of a *move* is therefore 1 unit.

The simple *move*(x) does not initiate a path compression from the processor to which x's forwarding address used to point.

For example, let S be the complete graph on 5 vertices (Figure 3.1). If R is currently located at processor 3 then a possible forwarding address tree in S is shown in Figure 3.2. Figure 3.3 illustrates the effect of a move(4) operation on the forwarding address tree of Figure 3.2. Figure 3.4 is a schematic representation of a move, emphasizing the fact that only two forwarding addresses are changed.

The three basic protocols differ in how the information learned in an *access* is used to update the forwarding address tree. We use the following three implementations of *access*. We will refer to each of the protocols by the name of the *access* operation it uses.

Lacc: In a lazy access operation, denoted *Lacc*(x), processor x dispatches a message along the path of forwarding addresses from x to the current location of R. The

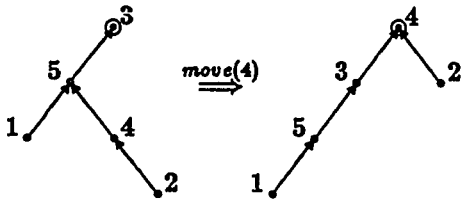

Figure 3.3: The mutation of the tree in Figure 3.2 by a move(4) command.

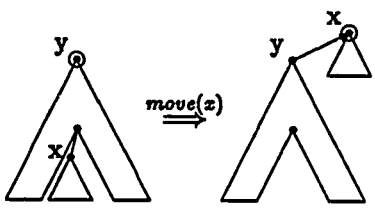

Figure 3.4: The operation move(x) represented schematically.

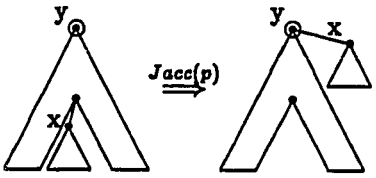

Figure 3.5: The operation $Jacc(p)$.

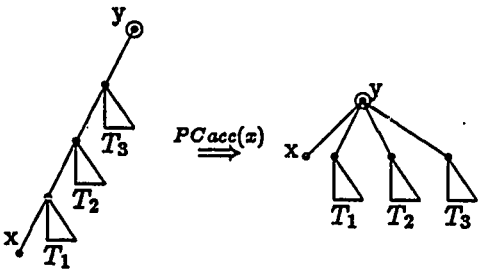

Figure 3.6: The effect of a $PCacc(x)$.

cost of the object finding part of operation is the length of this path. The messages sent on the edge between x and R's location are part of the access itself and are not charged to object finding. Processor x is lazy and ignores any information it receives in the course of the *access* about the current location of R. The forwarding address tree is therefore not modified by $Lacc(x)$.

Jacc: A successful access would usually entail x learning the current location of R. With no additional communication x can thus update its forwarding address for R, jumping from wherever it is in the forwarding address tree to being a child of the root. We denote this implementation by $Jacc(x)$. Figure 3.5 is a schematic representation of the effects of this operation on a forwarding address tree.

PCacc In a path compressing $PCacc(x)$ operation x updates its own forwarding address and sends a maintenance message containing the new address to the processor to which its old forwarding address pointed. This maintenance message is propagated along the original path as long as the new forwarding address replaces older ones. If there has been no concurrent activity in the system each processor on the path except for the last two update their forwarding addresses and the original path is completely compressed. Figure 3.6 illustrates the effect of a $PCacc(x)$ operation. Since messages are only sent along the original path the cost of a $PCacc$ is, like

the other operations, the length of the path.

Protocol	Lower Bound	Upper Bound	Average Case
Lacc	$a(N-1)$	$a(N-1)$	$\approx a\sqrt{\frac{\pi}{2}N}$
Jacc	$a\lfloor\frac{m(N-2)}{a}\rfloor$	$a(N-1)$, $a \leq m$ $a+m(N-1)$, $a > m$	$\approx a\sqrt{\frac{\pi m}{2(a+m)}N}$
PCacc	$a+a\lfloor\log_{(1+\lfloor a/m\rfloor)} N\rfloor$	$a+3a\log_{1+a/m} N$	

Table 3.1: Summary of the total cost of access operations when there are N processors and a total of a accesses and m moves. These are the asymptotic results for $a, m >> N$. The bounds for *PCacc* assume $a \geq m$.

Table 3.1 summarizes our results for the total cost of executing the *access* operations in a problem instance consisting of a *accesses* and m *moves* for each of these forwarding address protocols. The upper bound for *PCacc* quoted in the table is not our best, rather it is a weaker bound chosen because it is easily compared with the lower bound. Note that the bounds quoted are for asymptotically large problem instances.

We expect that most objects will be accessed more often than they move. Furthermore, as the number of processors "interested" in an object grows we expect that the ratio of accesses to moves will grow. Given these expectations our results are very encouraging. For both *Jacc* and *PCacc* as the ratio $\lambda = a/m$ grows the cost per access decreases. In fact, if λ grows at least as fast as ϵN for some constant ϵ, then the worst case cost per *access* for *Jacc* remains constant as N grows. In contrast, to ensure a constant ammortized cost per access, λ need grow only as fast as N^{δ} for any constant $\delta > 0$ if *PCacc* is used.

Notice that we have not quoted an analytic estimation of the average cost of using *PCacc*. The derivation of such an analytic result remains an open problem. Numerical results derived by simulation are discussed in Chapter 4.

3.2 Complexity of protocols using *Lacc* and *Jacc*.

Let N be the number of processors that appear in the problem instance. Let a be the number of *accesses* and m be the number of *moves*. Let $\lambda = a/m$ be the ratio of *accesses*

to *moves*. We are primarily concerned with analyzing behavior of the protocols when the a and m are large compared to N. We will therefore tend to ignore the effects of "start-up transients" and concentrate on cost per operation by analyzing the cost per operation for a fixed λ and N as a and m get very large.

The protocols using *Lacc* and *Jacc* exhibit poor worst case behavior. When *Lacc* is used a constructive lower bound is obtained by generating a problem instance which starts with fewer than N *moves* to build a forwarding address tree in the form of the linear graph on N vertices. Since *Lacc* does not modify the forwarding address tree every *Lacc* can originate at the vertex furthest from the root and thus cost $N-1$. Since there are $N-1$ edges in a tree on N vertices this is also the upper bound on the cost of a problem instance.

Lower bounds for the protocol using *Jacc* are also easy to construct. When $\lambda \leq 1$ there are enough *moves* in the problem instance to reconstruct the linear graph after each *Jacc*. The matching upper and lower bounds on the cost per *access* is therefore $N-1$ in this case.

In the case in which $\lambda > 1$ there are no longer enough *moves* in the problem instance to ensure that every *access* costs $N-1$. To derive our lower bound we first need the following Lemma.

Lemma 1 *Let T be any forwarding tree on N processors. There is a sequence of at most 2(N-1) moves that can construct T starting from any initial forwarding address tree configuration on N processors.*

Proof: We exhibit the recursive program *buildtree* (Figure 3.7) which, when given a specification of T and invoked with the root of T as its argument, executes a sequence of at most $2(N-1)$ *moves* and produces T from any starting configuration.

Buildtree(x) does a recursive traversal of the sub-tree of T rooted at x. Since the last thing that an invocation *buildtree*(x) does is a *move*(x) at either of lines A or C, when it returns it leaves the resource at processor x. The *move*(x) at line C builds the edge in T from y to x, its parent in T. If *buildtree* is initially invoked with the root of T

```
procedure buildtree(x: processor);
    begin
        if (x ∈ leaves(T)) then
            move(x)        { line A}
        else for (y ∈ { children of x in T}) do
            buildtree(y);    { line B }
            move(x)          { line C }
            end;
        return
    end;
```

Figure 3.7: Procedure buildtree creates the edges of a sub-tree in an arbitrary forwarding address tree.

as its argument, then it is called at line B exactly once for each of the other processors in V. Each edge of T is therefore created. Furthermore, since there are no further *moves* to y it can never be removed.

The number of moves that *buildtree* uses to construct T is equal to the number of leaves in T (line A) plus the number of edges (line C). In the worst case this is $2(N-1)$. □

Theorem 2 *Let* $\lambda' = \lfloor a/(m-2N+2) \rfloor$. *If* $\lambda' \geq 2$ *then there is a problem instance in which every access in the Jacc protocol costs* $\lfloor \frac{N-2}{\lambda'} \rfloor + 1$.

Proof: We explicitly construct such a problem instance. We begin by using an initial set of fewer than $2N-2$ *moves* to construct the forwarding address tree of height j illustrated in Figure 3.8. This tree has single vertices at depths 0 and 1. At depth i, where $2 \leq i \leq j$, there are exactly λ' vertices. Because this tree has N or fewer vertices we have $(j-1)\lambda' \leq N-2$ The construction can therefore be done with $j = \lfloor \frac{N-2}{\lambda'} \rfloor + 1$.

Starting in this configuration we can originate *Jacc*'s from the λ' vertices at depth j. This decreases their depth from j to 1, leaving a total of $\lambda' + 1$ vertices at depth 1. A *move* to one of these results in a forwarding address tree that is isomorphic to the starting configuration. This sequence of λ' *Jacc*'s followed by a single move can therefore be repeated indefinitely. □

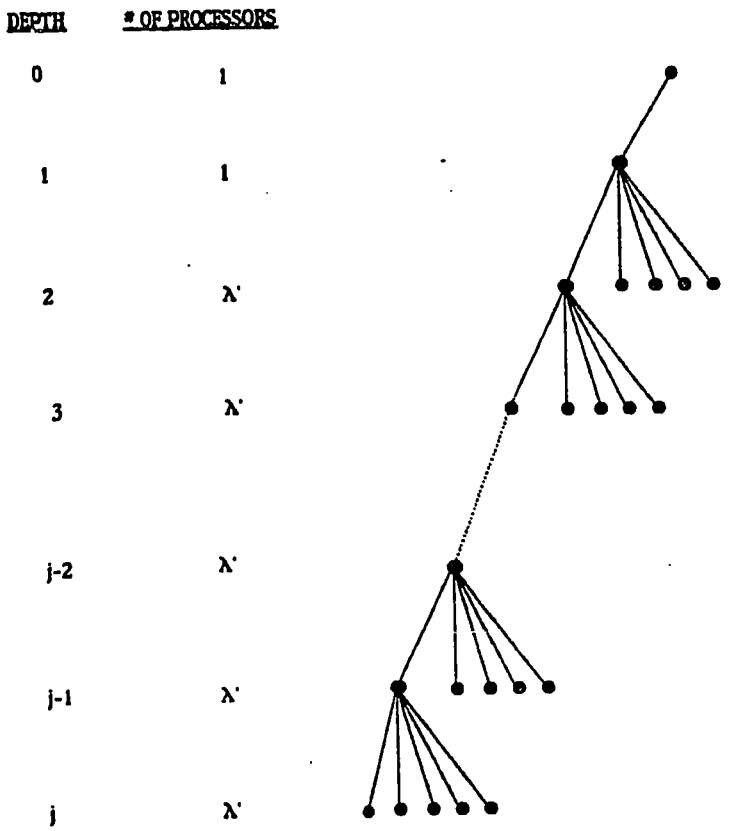

Figure 3.8: Constructive lower bound for the *Jacc* protocol. A forwarding address tree isomorphic to J is constructed from J using λ *accesses* and m *moves*.

The upper bound on the total cost of using the *Jacc* protocol is similar.

Theorem 3 *The total cost of executing the Jaccs in a problem instance with a accesses and m moves is less than or equal to* $a + m(N-1)$.

Proof: Let A be the set of *accesses* executed in the problem instance. Define c_i, where $i \in A$, to be the cost of executing *access* i. In a *move* the depth of a processor in a forwarding address tree can be increased by at most 1. Each move can therefore increase the sum of the processor depths by at most $N-1$.

Access i costs c_i and decreases the depth of its originator from c_i to 1. It therefore decreases the sum of the depths by at least $c_i - 1$. The initial forwarding address tree has the minimum possible sum of processor depths so the sum of the decreases cannot exceed the sum of the increases. We therefore have $m(N-1) \geq \sum_{i \in A}(c_i - 1)$, or $a + m(N-1) \geq \sum_{i \in A} c_i$. □

In contrast to these worst cases, applications using *ad hoc* centralization can achieve constant cost per *access*. In practice the actual performance will lie somewhere between these two extremes. We therefore turn to an average case analysis.

3.2.1 Using *Lacc* Generates Random Labeled Trees.

In analyzing the protocol using *Lacc* we need only consider the effect of *moves* on the forwarding address tree. We assume that the application that generates problem instances chooses the destination of each *move* from a uniform probability distribution. We call this the Uniform Random Move process, abbreviated URM. Using this assumption we analyze the protocol using *Lacc* as a Markov chain using each of the of the $M \stackrel{\text{def}}{=} N^{N-1}$ distinct rooted, labeled trees as a distinguishable state of the process. We show first that this chain is *irreducible* and *aperiodic* and therefore has a unique equilibrium state distribution. We then show that the chain is *doubly-stochastic*. It follows directly from this statement [Par67] that in equilibrium each of the trees is equiprobable with probability $1/M$.

The states are the rooted, labeled trees on N vertices and they are numbered from 1 to M using a standard scheme for enumerating them [Knu68].

Let p_{ij}^t be the probability of going from state j to state i in exactly t *moves*. By the definition of the URM process,

$$p_{ij}^1 = \begin{cases} 1/N, & \text{if } i \text{ one of the } N \text{ trees reachable from } j \text{ in a single } \textit{move}. \\ 0, & \text{otherwise.} \end{cases}$$

We chose to allow the null *move* because it does not change the analysis and allows the results to be expressed a little more compactly than otherwise. Because we allow the null *move* each tree is a possible successor to itself.

Let $f_i(t)$, $1 \leq i \leq M$, be the probability that after exactly t moves the state is the ith tree. Thus, $f_i(0) = 1$ when tree i is the initial configuration. The distribution after the $(t+1)$st move is

$$f_i(t+1) = \sum_{j=1}^{M} p_{ij}^1 f_j(t) \qquad 1 \leq i \leq M$$

In matrix notation, the state distribution at time t is the column vector $F(t) \stackrel{\text{def}}{=} [f_i(t), \ldots, f_M(t)]^\dagger$, the single step transition matrix is $P \stackrel{\text{def}}{=} (p_{ij}^1)$, and the t+1st transition is expressed as $F(t+1) = PF(t)$. To refer to a Markov chain we will use the name of its transition matrix.

We review some standard terminology used to describe Markov chains [Par67]. State i is *reachable* from state j if and only if there is some t such that $p_{ij}^t > 0$. A Markov chain is *irreducible* if and only if any state is reachable any other state. The *period* of a state s is the greatest common divisor of the set of integers $\{t \mid p_{ss}^t > 0\}$. A Markov chain is *aperiodic* if and only if all of its states have period 1. In an irreducible Markov chain any state having period 1 implies that all states have period 1. A Markov chain is said to be *doubly-stochastic* if and only if the sum of the conditional probabilities for entering each state sum to 1.

Lemma 4 *Markov chain P is* irreducible.

Proof: The procedure *buildtree* is an explicit contstruction for transforming any forwarding address tree into any other. P is therefore irreducible. □

Lemma 5 *P is* aperiodic.

Proof: We allow the null move and therefore $p_{ss}^t > 0$ for all $t \geq 1$. Since the greatest common divisor of 1 and any integer is 1 the chain is aperiodic. □

If we disallow the null *move* then for $N = 2$ the chain becomes periodic because there are only two possible states. For all $N \geq 3$ it is an easy exercise to show that it is aperiodic by displaying cycles of length 2 and 5 for states with 2 or more children of the root and of length 2 and 3 for states in which the root has only one child.

The two conditions of irreducibility and aperiodicity ensure that the chain has a unique *steady state* or *equilibrium* distribution Π such that $\Pi = P\Pi$.

Let x be a vertex and T be a forwarding address tree. Let $subtree(x,T)$ denote the sub-tree of T rooted at x and let $|subtree(x,T)|$ denote the number of vertices in $subtree(x,T)$ (including x). The set of children of x in T is denoted by $children(x,T)$.

Lemma 6 *P is* doubly-stochastic.

Proof: All the non-zero conditional probabilities p_{ij}^1 are equal to $1/N$. We now show that each state (tree) has N predecessors and therefore that the sum of the conditional probabilities for entering that state sum to 1.

Let the tree generated after the tth step be denoted by T_t and let the root of T_t be denoted by $root_t$. We will count the number of trees that could have been T_{t-1}, T_t's predecessor.

Case 1: $T_{t-1} = T_t$. That is, T_t was its own predecessor. There is exactly one way that this occurs.

Case 2: $T_{t-1} \neq T_t$. Recall that a $move(x)$ (where $x = root_t$) mutates tree T_{t-1} into T_t by disconnecting the sub-tree rooted at x from T_{t-1}, and making $root_{t-1}$ a child of x. (See Figure 3.4.) All other edges of T_{t-1} are left undisturbed. We will count the number of ways that this could have been done to create T_t.

Suppose that $y \in children(x,T_t)$, was $root_{t-1}$. This implies that all the other children of x in T_t were also children of x in T_{t-1}. The parent of x in T_{t-1} therefore had to have been one of the vertices in the sub-tree of T_t rooted at y. The number of possible predecessors to T_t in which y was the root is therefore $|subtree(y,T_t)|$.

Since any vertex in $children(x, T_t)$ could have been $root_{t-1}$ and since each of the $N-1$ vertices other than x is in exactly one of the sub-trees rooted at $children(x, T_t)$, the number of possible predecessors of T_t not rooted at x is $N-1$.

The total number of possible predecessors of T_t is N. Since all of the non-zero entries of P are $1/N$, the sum of conditional probabilities leading to each tree is 1. P is therefore *doubly-stochastic.* □

Theorem 7 *In the equilibrium distribution of P each of the $M \stackrel{\text{def}}{=} N^{N-1}$ rooted, labeled trees with N vertices occurs with equal probability.*

Proof: The lemmas prove that the P is *irreducible, aperiodic,* and *doubly-stochastic.* The first two properties guarantee that P has a unique equilibrium distribution. The last property means that it is verifiable by inspection that the vector $\Pi = [1/M, \ldots, 1/M]^\dagger$ satisfies $\Pi = P\Pi$ and is therefore the equilibrium distribution. □

3.2.2 The Distribution of Vertex Depths in Rooted Labeled Trees.

In this section we again analyze the protocol using *Lacc* as a Markov chain, but this time using a different state space. The *depth* of a vertex in a rooted tree is the number of edges in the path from that vertex to the root. We choose an arbitrary vertex x and let the state of the process at time t be the *depth* of x in the forwarding tree created by the tth step of the URM process. When there are N vertices the depth of x can range from 0 to $N-1$. In Figure 3.9 the depth of vertex x is k.

Assume that after step $t-1$ that the depth of x is k. What are the conditional probabilities that x will be at depth j after the tth step of the URM process? There are three distinct cases:

$0 < j \leq k$: This can occur only if the tth step is a $move(y)$ where y is the unique vertex located j edges above x on the path to the root. The probability of this occuring is $1/N$.

$j = k+1$: Whenever the tth move is to a vertex not on the path from x to the root the depth of x is increased by 1. The probability of this occuring is $(N-(k+1))/N$.

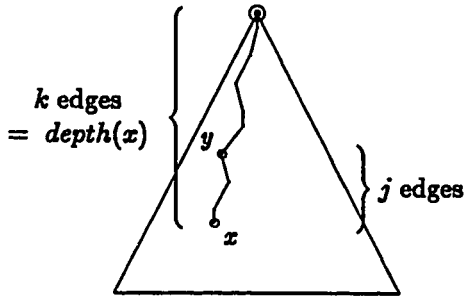

Figure 3.9: Vertex x at depth k will be at depth j if a $move(y)$ is executed.

$$Q = \frac{1}{N}\begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ N-1 & 1 & 1 & \dots & \vdots \\ 0 & N-2 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & 1 & 1 \end{pmatrix}$$

Figure 3.10: The transition matrix Q using $depth(x)$ as the state.

$k+1 < j \leq N-1$: There is no way for the depth of x to increase by more than 1 in a single step so the probability of this occuring is 0.

Since the conditional transition probabilities depend only upon the depth of x, the URM process analyzed with this state space is again a Markov chain. The single step transition matrix Q is illustrated in Figure 3.10.

As in the previous analysis, the chain Q is *irreducible* and *aperiodic* and therefore has a unique equilibrium distribution Θ such that $\Theta = Q\Theta$, where $\Theta = [\theta_0, \dots, \theta_{N-1}]^\dagger$ and each θ_i is the probability that x will be at depth i. We can solve the system $(Q-I)\Theta = 0$ by noting that the first equation of the system normalizes the θ's to sum to 1 and each of the remaining equations is of the form

$$(N-i)\theta_{i-1} + (1-N)\theta_i + \sum_{j=i+1}^{N-1} \theta_j = 0 \qquad 1 \leq i \leq N-1, \tag{3.1}$$

recursively defining θ_{i-1} in terms of the θ's with larger indices. It is easily verified by substitution into Equation 3.1 that

$$\theta_{N-k-1} = \left(\frac{(N-1)!}{N^N}\right)(N-k)\frac{N^k}{k!} \qquad 0 \leq k \leq N-1 \tag{3.2}$$

is the closed form solution. This can also be rewritten as

$$\theta_i = \frac{(N-1)!}{(N-i-1)!}\frac{i+1}{N^{i+1}} \qquad 0 \leq i \leq N-1.$$

The expected value of the depth of a vertex is computed by evaluating $\bar{d} = \sum(N - k - 1)\theta_{N-k-1}$ using Equation 3.2. Simple manipulations of the right hand side yield the form

$$\bar{d} = \frac{N!}{N^N} \sum_{j=0}^{N-2} \frac{N^j}{j!}.$$

This formula is the power series for e^N truncated to $N-1$ terms and normalized. The function $Q(n)$ [Knu68] (pages 112-113, equations (2) and (4)) is the same formula but truncated to N terms. Noting that the Nth term when normalized is equal to 1, we use Knuth's asymptotic approximation of $Q(n)$ (page 117) to obtain the result

$$\bar{d} = \sqrt{\frac{\pi N}{2}} - \frac{4}{3} + \frac{1}{12}\sqrt{\frac{\pi}{2N}} - \frac{91}{540N} + \frac{1}{288}\sqrt{\frac{\pi}{2N^3}} + O(N^{-2}) \tag{3.3}$$

By examining the ratio d_{i+1}/d_i the mode of the distribution of vertex depths is found to be at $\sqrt{N+1}$.

The *height* of a tree is the maximum depth of any vertex in that tree. Renyi and Szekeres [RS67] have shown that the expected value for the height of a rooted labeled tree is asymptotically $\sqrt{2\pi N}$.

Meir and Moon [MM70, MM78] have published the distribution

$$\theta_i = \frac{N!}{(N-1)(N-i-1)!} \frac{i+1}{N^{i+1}}$$

for the probability that a vertex in a random rooted, labeled tree will be at depth i. Each value in this distribution differs from our result by a factor of $N/(N-1)$. This is explained by noting that they neglect the event that x will be the root of the tree and have depth 0. (This occurs with probability $1/N$.) They also derived estimates for the expected depth of a vertex and our analysis improves upon their results.

3.2.3 Average Case Analysis of the Cost of Using *Jacc*.

We now consider the more general case in which at each step the application chooses with probability α to execute an $Jacc(y)$ and with probability $1-\alpha$ to execute a $move(y)$. In both cases processor y is chosen from a uniform distribution. Once again, we will use the depth of node x as the state of the process.

$$A = \frac{1}{N}\begin{pmatrix} N & 0 & 0 & \dots & 0 \\ 0 & N & 1 & \dots & 1 \\ 0 & 0 & N-1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 1 \\ 0 & \dots & \dots & 0 & 2 \end{pmatrix}$$

Figure 3.11: The transition matrix A for a *Jacc* chosen from a uniform distribution.

The parameter λ used previously is the ratio of accesses to moves in a problem instance. We will relate α and λ using the approximations $\lambda \approx \alpha/(1-\alpha)$ and $\alpha \approx \lambda/(1+\lambda)$.

The effect of a single $Jacc(x)$ on a forwarding address tree is illustrated in Figure 3.5. If node x is at the root of the tree then any *Jacc* will leave it there. Otherwise, if x is at depth $i \neq 0$ then for $0 < j < i$ the probability that a randomly chosen *Jacc* will leave x at depth j is $1/N$ and the probability that the depth remains unchanged is $(N-i)/N$. Figure 3.11 is the transition probability matrix A for an single *Jacc* operation. A has two absorbing states corresponding to x being the root and being a child of the root.

Since the application chooses to make a *move* with probability $1-\alpha$ and an *Jacc* with probability α, its transition probability matrix is $C(\alpha) = (1-\alpha)Q + \alpha A$ where $0 \leq \alpha < 1$. As long as $\alpha < 1$ (that is, the probability of a *move* is non-zero) this process is irreducible and aperiodic. It therefore has an equilibrium solution $\Phi(\alpha) = [\phi_0(\alpha), \dots, \phi_{N-1}(\alpha)]^\dagger$ that satisfies the system of equations $(C(\alpha) - I)\Phi(\alpha) = 0$. As in the previous section, the first equation expresses a normalization condition and the rest recursively define each of the components of $\Phi(\alpha)$ in terms of the higher indexed components.

$$(1-\alpha)(N-i)\phi_{i-1}(\alpha) + [\alpha(N-i) + 1 - N]\,\phi_i(\alpha) + \sum_{j=i+1}^{N-1} \phi_j(\alpha) = 0 \qquad (3.4)$$

When $\alpha = 0$ Equation 3.4 reduces to Equation 3.1 so $\Phi(0) = \Theta$. We have not found a closed form solution when $\alpha \neq 0$. Since $C(\alpha) - I$ is almost upper triangular, numerical solutions for fixed values of N and α are easily computed. The expected value of the

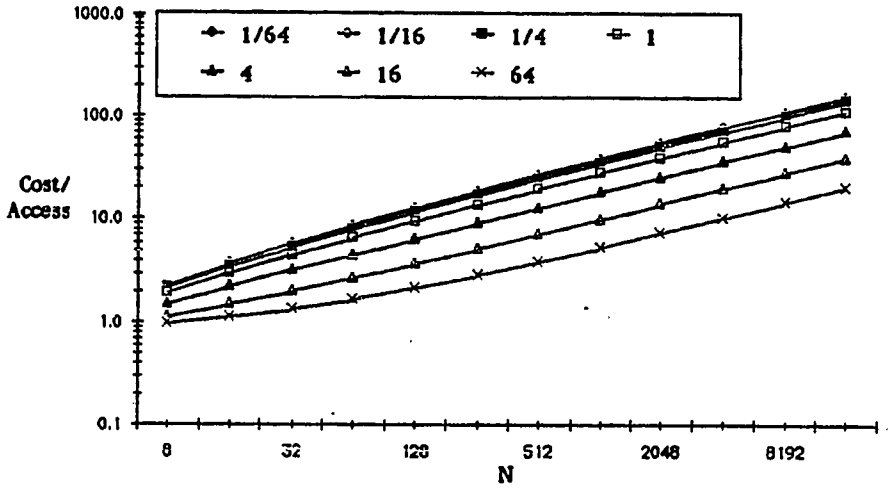

Figure 3.12: The expected cost of using *Jacc*. Numerical solutions for the expected depth of a vertex ($\bar{d} = E(d)$) as a function of N and α. Each curve represents a different choice of α.

depth of a vertex for several values of N and α are plotted in Figure 3.12.

If at some time node x is at depth j then the expected depth after the next step, the *conditional expectation* of the depth, is defined as

$$r_j(\alpha) = \sum_{i=0}^{N-1} i c_{ij}(\alpha) \qquad 0 \le j \le N-1.$$

This evaluates to

$$r_j(\alpha) = -\frac{j^2}{2N} + \left(\frac{2\alpha - 1}{2N} + 1\right) i + 1 - \alpha \qquad 0 \le j \le N-1.$$

The point $\hat{j}(\alpha)$ such that $r_{\hat{j}}(\alpha) = \hat{j}(\alpha)$ is a convergence or regression point of $C(\alpha)$ in the sense that if the depth d of x is larger than $\hat{j}(\alpha)$ then the expected depth after the next step will be smaller than d and conversely if d is smaller than $\hat{j}(\alpha)$ then the expected depth after a step will be larger than d. Asymptotically, $\hat{j}(\alpha) \approx \sqrt{2(1-\alpha)N}$. Although $\hat{j}(\alpha)$ bears no direct relation to the mean of $\Phi(\alpha)$, it is in a sense a measure of the central tendency of the distribution. Taking a hint from the functional form of $\hat{j}(\alpha)$ we observe that $\bar{d}(\alpha) \approx \sqrt{\frac{\pi}{2}(1-\alpha)N}$ is a very good approximation to the numerically computed expected depth of a vertex when α is in the range of the values used in Figure 3.12.

Using this approximation as the expected cost of an *Jacc* the average cost per operation of a stream of *moves* and *Jacces* is

$$1 - \alpha + \alpha\sqrt{\frac{\pi}{2}(1-\alpha)N}. \tag{3.5}$$

For fixed N this is at its maximum when $\alpha \approx 2/3$.

We leave as open problems the derivations of a closed form for $\Phi(\alpha)$ and the computation of its mean.

Doyle and Rivest [DR76] analyzed the set union algorithm corresponding to our *Lacc* protocol and derived an $O(1)$ average cost bound for a certain restricted set of assumptions. Our results of this section indicate that using our uniform choice assumptions that neither the *Lacc* nor the *Jacc* protocols exhibit constant cost per operation behavior. We leave as an open area of research the problem of deciding which sets of average case assumptions result in constant cost results and which do not.

3.3 Worst Case Bounds Using *PCacc*.

In this section we analyze the worst-case performance of the protocol using *PCacc*. As in the case of the other two protocols an adversary can use $N-1$ moves to build a linear chain of forwarding addresses. The worst case cost of any single *PCacc* is therefore $N-1$. Not all *PCacc*'s can cost this much so we will analyze the total cost of executing an entire problem instance, the entire sequence of *move*'s and *PCacc*'s that are executed in R's lifetime.

We characterize a problem instance with three parameters: N, the cardinality of the set of processors that R visits or that access it; m, the number of *move*(x) operations in the problem instance; and a, the number of *PCacc*(x) operations. Let $C(N, a, m)$ denote the worst case cost of running an instance with those parameters. Because we are interested in the practical application of the protocol we attempt to make our analyses as concrete as possible.

We will concentrate on the case in which objects are accessed remotely at least as often as they are moved. We are also concerned with the long term behavior of the

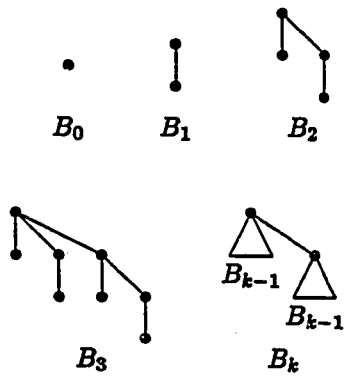

Figure 3.13: Binomial trees.

protocol so we will concentrate on the cases in which a and m are both much larger than N.

3.3.1 Lower Bounds.

The lower bounds on the worst case performance of the protocol are obtained by exhibiting a method for constructing arbitrarily long sequences of *move*'s and *PCacc*'s in which the *PCacc*'s are relatively expensive. The general idea is to use an initial sequence of between $3N/2$ and $2(N-1)$ *move*'s to construct a forwarding address tree T with the "self-reproduction" property that starting with T there is a sequence of relatively "expensive" *PCacc*'s followed by a single *move* that leaves a forwarding address tree isomorphic to T. This is the same technique we used for constructing the lower bound on the cost of the *Jacc* protocol. We adapt a construction due to Fischer [Fis72] and refined by Tarjan and Van Leeuwen [TvL84]. We begin by discussing the adaptation of Fischer's construction in detail.

The construction is based on the remarkable properties of a recursively defined family of trees called ***binomial trees.*** The binomial tree B_0 is an isolated vertex and $B_i, \quad i > 0$ is constructed recursively by taking two copies of B_{i-1} and joining them by making the root of one copy a child of the root of the other. See Figure 3.13. The binomial tree

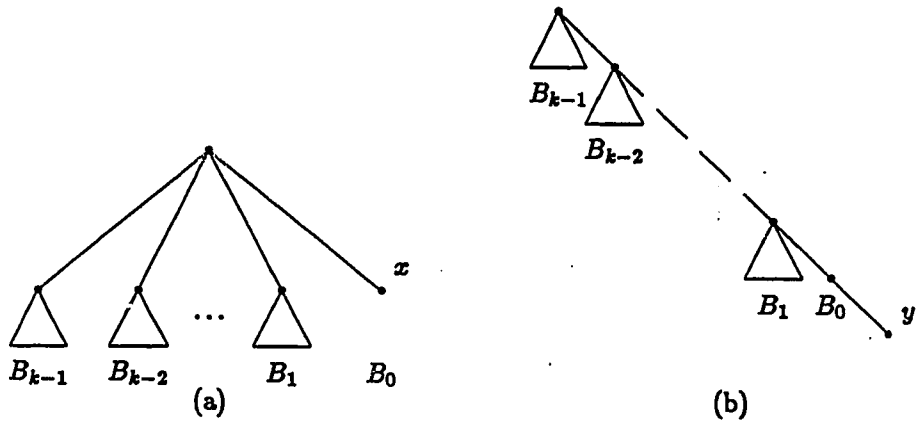

Figure 3.14: Horizontal (a) and vertical (b) decompositions of B_k. Vertex x is the *grip* and y is the *handle*.

B_k contains 2^k vertices of which exactly half are leaves. In B_k the number of vertices at depth j is the binomial coefficient $\binom{k}{j}$. The height of B_k is exactly k and there is exactly one vertex at that depth, called the *handle*. We call the single leaf at depth 1 B_k's *grip*.

Because of its recursive definition, a binomial tree of order k contains many disjoint copies of binomial trees of lesser order. In particular, Fischer noted two particularly useful decompositions of B_k (Figure 3.14). The horizontal decomposition of B_k *(a)* is obtained by distinguishing the sub-trees rooted at children of B_k's root. Similarly, the vertical unrolling is obtained by counting the sub-trees rooted on the path to the handle. Using these decompositions he showed that if one starts with a tree containing a copy of B_k embedded anywhere but at the root and do a path compression beginning at its handle then the resulting tree will have a copy of B_k embedded at its root. Furthermore, the handle of the original B_k will be a child of the root but will not be in the embedded copy of B_k. Binomial trees are therefore in a sense "self-reproducing" under path-compressing operations. Figure 3.15 illustrates a forwarding address tree that is reproduced by the pair of operations consisting of a *PCacc* of cost $k+1$ originating at the current handle of the embedded B_k tree followed by a *move* to that processor. The total number of processors used is 2^k+1. To obtain the self-reproducing property with one fewer vertex

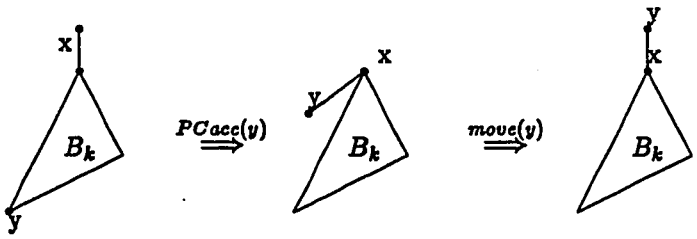

Figure 3.15: The self-reproduction of B_k.

we begin by transforming a B_k tree into what we call a R_k tree by executing a *move* to its grip. The R_k tree therefore has $2^{k-1}-1$ leaves and its handle is at depth $k+1$. Executing a *PCacc* originating at the handle converts the R_k tree into a new B_k tree with the handle of the original tree becomming the grip of the new one.

Theorem 8 *Let $m \geq 3(N-1)/2+a-1$. There is a sequence of m moves and a PCaccs on N processors such that every PCacc costs at least $\lfloor \log_2 N \rfloor + 1$.*

Proof: By Lemma 1 an adversary can construct a copy of R_k where $k = \lfloor \log_2 N \rfloor$ in a forwarding address tree using at most $3(N-1)/2-1$ *moves*. Each *PCacc* is from R_k's handle, costing $k+1$ and producing a copy of B_k. Of the remaining *moves* $a-1$ are used to transform the B_k back into a copy of R_k. The total cost of executing the problem instance is therefore $m + a(\lfloor \log_2 N \rfloor + 1)$. □

The self-reproduction sequence for binomial trees consists of a single *PCacc* followed by a *move*. In order to derive a lower bound for the case in which after the initial construction phase there are $k = \lfloor \lambda \rfloor$ times more *PCaccs* than *moves* it is sufficient to find a family of trees with a self-reproduction sequence consisting of k *PCaccs* followed by a single *move*. Tarjan and van Leeuwen [TvL84] defined T_j^k, a family of trees with exactly this property. The cost of each path compression in the self-reproducing sequence is i where $j = ik$. Let m' be the number of *move*'s remaining after the the initial construction phase. T_j^k contains fewer than $(k+1)^{i-1}$ nodes so choosing $i = \lfloor \log_{k+1} N \rfloor + 1$ and $k = \lfloor \frac{a}{m'} \rfloor$ yields a j such that $|T_j^k| \leq N$. The cost of constructing T_j^k is at most $2N-1$

and may be as little as $3N/2$. With this choice of the parameters the total cost of the *PCacc*'s is at least $a\left[\log_{(1+a/m')} N + 1\right]$.

Theorem 9 *Define the quantity m' such that $m-2N \leq m' \leq m-3N/2$. This represents the number of move operations left after the startup phase. If $m' \geq 0$ and $a \geq m'$, then $C(N,a,m) \geq m + a + a\lfloor\log_{(1+\lfloor a/m' \rfloor)} N\rfloor$.*

Proof: (Sketch) The Tarjan and van Leeuwen trees are similar to binomial trees and they are used in a similar way. □

3.3.2 Upper Bounds.

To derive a worst-case upper bound on the cost of executing the resource finding protocol using *PCacc* on a problem instance with N vertices, m *moves*, and a *PCaccs* we refine a technique attributed to Paterson [FM77, Yao85]. The idea is to put an upper bound on the total cost attributable to expensive *accesses*, those that cost more than $\rho \ln N$ where ρ is a parameter of the analysis and may be chosen as a function of N, m, and a in order to optimize the analysis. The bound on the total cost is obtained by adding this to the possible cost of all of the inexpensive *accesses*.

Let x be a vertex and define σ_x, the weight of x, to be the number of vertices in the sub-tree rooted at x. We define the quantity $H = \sum_{x \in V} \rho \ln \sigma_x$ to be the *entropy* of the the forwarding address tree. Although it is not the same as the entropy defined in information theory, it is a measure of the disorder of the system. H is reduced by *PCacc* operations and it can be increased by *moves*.

When all of the processors know R's location, i.e. the height of the tree is 1, then H takes its minimum value of $\rho \ln N$. The execution of the protocol starts in this minimal entropy configuration. All other configurations of the forwarding address tree have a larger value of H, but less than $\rho N \ln N$.

Lemma 10 *The operation move(x) can increase H by at most $\rho \ln N$.*

Proof: The only vertex whose weight is increased by *move*(x) is x. Since x's weight can

be increased by at most N this can increase H by at most $\rho \ln N$ by the definition of H. The decreases to the weights of other vertices can only decrease H. □

Define ΔH_l to be the amount by which H is decreased by a *PCacc* that uses a path of $l+2$ processors and whose cost is therefore $l+1$.

Lemma 11 *$\Delta H_l \geq l\rho \ln \frac{\beta}{\beta-1}$ for a value of β depending on l, where $0 < \ln \beta < \frac{\ln N}{l}$.*

Proof: (Sketch, after Yao [Yao85]) Let T be the tree before the path compression and let T' be the tree afterwards. Let the vertices along the path be $v_0, v_1, \ldots, v_l, v_{l+1}$, where v_{l+1} is the root of T. The change in entropy is expressed by

$$H(T) - H(T') = \rho \sum_{i=1}^{l} (\ln \sigma_{v_i} - \ln(\sigma_{v_i} - \sigma_{v_{i-1}})) .$$

The weights are defined with respect to T. Each term in the summation is the contribution due to the change in the weight of v_i. The weights are constrained to be monotonic, i.e. $1 \leq \sigma_{v_0} < \sigma_{v_1} < \ldots < \sigma_{v_{l+1}}$. Under this constraint the sum is minimized when the weights are minimized when they are in a geometric sequence, $\sigma_{v_i} = \alpha\beta^i$, $\alpha \geq 1$. The result follows by straighforward manipulation. □

Theorem 12 *Let $C(N, a, m)$ denote the total cost of executing a sequence of m moves and a PCacc's with N processors. Then*

$$C(N, a, m) < m + a(1 + \rho \ln N) + \frac{m \ln N}{\ln \frac{e^{1/\rho}}{e^{1/\rho}-1}}, \tag{3.6}$$

where $\rho > 0$ is a parameter of the analysis and may be chosen based on N, m, and a to minimize the right side of the inequality.

Proof: Let c be the number of "cheap" *PCacc* operations where "cheap" is defined as costing no more than $1 + \rho \ln N$. There is thus a set E of $a - c$ "expensive" operations each of which costs more than $1 + \rho \ln N$. We consider separately the costs accrued by *moves*, the c "cheap" *PCaccs* and the $a - c$ "expensive" *PCaccs*.

Each *move* is charged one cost unit so the total charge to *moves* is m.

The c "cheap" *PCacces* each cost less than $1+\rho\ln N$ so their contribution to the total is bounded by $c\cdot(1+\rho\ln N)$.

The cost accrued by i, an "expensive" *PCacc*, can be written as $1+l_i$ where $l_i > \rho\ln N$. The total cost accrued by the "expensive" operations is therefore $\sum_{i\in E}(1+l_i)$ or $a-c+\sum_{i\in E} l_i$.

Let $l = l_i$. By Lemma 11, $\Delta H_l \geq l\rho\ln\frac{\beta}{\beta-1}$, where $0 < \rho\ln\beta < \frac{\rho\ln N}{l}$. Since $l > \rho\ln N$ by the definition of "expensive", $\rho\ln\beta < 1$ and therefore $1 < \beta < e^{1/\rho}$. Thus, $\rho\ln\frac{\beta}{\beta-1} > \rho\ln\frac{e^{1/\rho}}{e^{1/\rho}-1}$ and $\Delta H_l > l\rho\ln\frac{e^{1/\rho}}{e^{1/\rho}-1}$.

By Lemma 10, the total entropy created by executing m *moves* is no greater than $m\rho\ln N$. Since the *PCacces* cannot reduce H below its initial value we have

$$\rho\ln\frac{e^{1/\rho}}{e^{1/\rho}-1}\sum_{i\in E} l_i < \sum_{i\in E}\Delta H_{l_i} \leq m\rho\ln N.$$

The total cost accrued by the "expensive" *PCacces* is therefore less than

$$a-c+\frac{m\ln N}{\ln\frac{e^{1/\rho}}{e^{1/\rho}-1}}.$$

Adding the upper bounds on the costs accrued by each type of operation gives us the desired result. □

Making the substitutions $m = a/\lambda$ and noting that $\ln(z)-\ln(z-1) > \frac{d}{dz}ln(z) = 1/z$ we obtain the weaker but simpler bound,

$$C(N,a,m) < m+a+\left(\rho+\frac{e^{1/\rho}}{\lambda}\right)a\ln N, \tag{3.7}$$

which is optimized by choosing ρ such that $\lambda\rho^2 = e^{1/\rho}$. Define $g(\lambda,\rho) = \left(\rho+\frac{e^{1/\rho}}{\lambda}\right)$. If we set $\rho = 2/\ln(1+\lambda)$ we obtain $g(\lambda,\rho) = 2/\ln(1+\lambda)+(\lambda+1)^{1/2}/\lambda$. Thus,

$$C(N,a,m) < m+a+(2a\log_{1+\lambda}N)+\left(\frac{(\lambda+1)^{1/2}}{\lambda}\right)a\ln N. \tag{3.8}$$

This compares favorably with the lower bound. For $\lambda \geq 1$ the last term of 3.8 is less than $a\log_{1+\lambda}N$. For moderately large λ, such as $\lambda > \ln^2 N$, this term becomes less than a and upper bound on the ammortized cost per access is thus about twice the corresponding lower bound.

The restricted case in which R never revisits a processor can be analyzed by doing a direct reduction of an execution of the *PCacc* protocol to an execution of the algorithm for the set union problem that executes m "naive" ***link*** operations and a path compressing ***find***'s. If $a < m$ an upper bound of $m + a\,(1 + 2\lceil \log_2 m \rceil)$ applies [TvL84] . When $a \geq m$ our improvement of Paterson's technique applied to this set union algorithm results in an upper bound of $m + a + 3a \log_{1+a/m} m$, $a/m \geq 1$, improving in several ways the upper bound for this algorithm reported in [TvL84].

3.3.3 The Average Case Cost of Using *PCacc*.

For the protocol using *PCacc* we do not know of any analytically derived average case bounds that are any better than our worst case bounds. The simulations discussed in Chapter 4 indicate that when $a > m$ and the *move*'s and *PCacc*s are chosen from a uniform distribution that for $\lambda \geq 1$ the average case cost is at least one half of the lower bound.

3.3.4 Tree Maintenance as a Part of a *move*.

The protocols that we have analyzed have used the simplest legal implementation of a *move*. A reasonable conjecture is that we could make one or more of our protocols more efficient by somehow improving the *move* operation.

Our discussion of *ad hoc* centralization suggests one way of doing this. If the system were allowed to break the symmetry in the roles played by processors by designating a "home" processor h then it could double the cost of a *move* by transforming every occurence of "*move*(x)" in a problem instance to the pair "*move*(h); *move*(x)". If *Lacc* is used with this strategy then the maximum possible length of a forwarding address chain is 2.

If either *Jacc* or *PCacc* are used then there is no such guarantee. Long paths can still be constructed in the worst case. If R is at processor y before a *move* then the lower bound constructions can be modified by adding an *access*(y) after each *move*. If h is a special processor that otherwise does not participate in the problem instance,

then the combined effects of the system and the adversary are to change "*move*(x)" into "*move*(h); *move*(x); *access*(y);". The original *accesses* cost the same as they did in the original construction. The m new *moves* each have unit cost and the m new *accesses* each cost 2 units.

Another possible strategy is use a path compressing *move*, *PCmove*(x), that compresses the path from x's parent at the start of the operation to x, the new root of the tree. Unfortunately, this augmentation has only a minor effect on the lower bounds. All of the *moves* in the lower bound constructions for *Jacc* and *PCacc* are to processors that are at that moment children of root of the forwarding address tree and path compressions would have no effect at all. To prevent the attempted path compressions from having any effect during the initial phase of the construction all that is needed is modify the *buildtree* program in Figure 3.7 so that the *move*(a) in "line A" is also executed for the interior nodes of the tree under construction. This has the effect of increasing the cost of constructing any tree to $2N$ regardless of the number of leaves. Using *PCmove* can therefore produce at most a minimal worst case improvement in performance when $a \geq m$.

The average case analyses of these strategies are left for future work.

3.4 Refining the Definitition of N

Our analyses have taken N to be the number of processors "interested in R". This includes all processors that R can visit and those that can refer to it over its lifetime. We can refine our analyses by improving the definition of N.

For the purposes of our worst case analysis of *PCacc* N could have been restricted to be the set of processors that R visits. This is because processors that can refer to R but which are never visited will always be leaves of the forwarding address tree. An access originating at one of these leaves will traverse one more edge than one originating at the interior vertex to which it is attached. Furthermore, the same path compression occurs on the interior vertices. Thus, by redefining N we add at most a single unit of cost per access. If R visits much fewer processors than access it this can improve the

analysis substantially.

Similar improvements are easily obtained for the other analyses. Restricting N to be the number of processors visited decreases the depths of the forwarding address trees that are created by the processes generating the problem instances.

Using the *Lacc* protocol a processor never visited by R always keeps its original forwarding address. It's average depth in the tree will thus be exactly one more than that of the processor it points to.

The analyses for *Jacc* is not so easy. They must be parameterized not only in terms of the number of visited processors and the number of operations of each type that they originate, but also in terms of the number of never visited processors and the number and pattern of their accesses. Using the *Jacc* protocol an *access* originating at a leaf of the tree does not modify the interior of that tree. An *access* originating at a never visited processor does not therefore shorten the paths potentially traversed by other processors. The costs of the accesses originated by the visited processors are therefore independent of the never visited processors and their behavior. The cost of an access originated by a never visited processor can be at most one unit more that the cost of a potential access from a visited one. This is true both of the worst case and the average case analyses.

3.5 The Effects of Concurrency Upon Cost.

Thus far we have analyzed the protocols as though the operations in a problem instance were executed serially. In practice, however, there will be some amount of concurrency. If the processors keep enough state to know that they expect to receive an updated forwarding address for an object, then they can delay other *access* requests until that information is received. This will occur when the processor has originated a not yet completed *Jacc* or is on the path taken by a *PCacc*. The effect of this will be to possibly increase the delay seen by the later operations, but to keep the same costs as would be seen in a serial execution of the protocol.

If the processors do not maintain this extra state then the effect of concurrency will be to increase overall costs because there will be access messages traveling over paths

that would have already been shortened, either through a *Jacc* or a *PCacc*. We have no direct analytic estimate of the magnitude of this effect.

The path compressing phase of a *PCacc* is a maintenance operation on the forwarding address tree and is not needed to ensure the correctness of the protocol. We might be tempted to reduce congestion during peak hours on the system by making path compression a low priority activity. This has the effect of increasing the effective concurrency of the protocol. The effect to deferring all path compressions until a period of low system utilization would be to turn all *PCacc*'s into a *Jacc*'s during the period of high utilization. If the costs of using *Jacc* are acceptable for a short period of time then delaying path compression could be a reasonable idea. In general, however, this appears to be a poor strategy.

3.6 Discussion and Concluding Comments.

The worst case average cost per operation when we use *PCacc* can grow as $O(\log_2 N)$ for an arbitrary application. We know of no way of improving on this without introducing some form of centralization. We speculate that this is the cost penalty of using our particular definitions of autonomy and decentralization.

In their study of the cost of using a rendezvous mechanism Mullender and Vitányi [MV85] defined decentralization in terms of the frequency with which each processor can serve as a rendezvous between another pair of processors. In the fully decentralized case this frequency distribution is uniform across all processors. Using this model they proved an $\Omega(\sqrt{N})$ messages lower bound for the cost of establishing a rendezvous, where N is the number of processors in the system. While some part of the difference between the costs of using forwarding addresses versus rendezvous may in fact be attributable to the mechanisms themselves, they cannot really be compared unless they are analyzed under the same model. We leave as an open problem the development of a model general enough to subsume both our work and theirs.

An unexplored facit of the analysis is the study of the actual behavior of large, decentralized, object–oriented systems in which there are large numbers of small, mobile

objects. Such systems do not as yet exist. Many applications on decentralized systems will themselves be inherently centralized and can use centralized generalizations of forwarding addresses. We speculate that the locality properties of real applications would introduce forms of *ad hoc* centralization that would result in actual performace much better than our analyses predict.

One such locality property may be an inverse relation between the number of processors that may want to access an object and the rate at which it moves. As we saw above, if $\lambda = a/m$ grows at least as fast as a fixed fraction of N, the number of processors that can name an object, then the ammortized cost of an access using the *Jacc* protocol will be constant. If λ grows only as fast as a fixed power of N', where N' is the number of processors the object *visits*, then using *PCacc* will result in a constant cost per operation.

Our cost measure is crude. By choosing to count the network edges used in an operation we are attempting to approximate in one measure both the number of messages sent and the number of forwarding addresses that need to be looked up in the local databases of the processors. Without detailed knowledge of the actual implementation of some of the low level operations and of the underlying hardware, we do not know which of these will be more important. For instance, the cost of looking up a forwarding address may be dominated by the cost of accessing secondary storage. This can be several orders of magnitude greater than the cost of sending a single packet message in a local area network.

We used the number of edges traversed by an *access* request as the measure of its cost. We did not count the edges that may be created by that operation. In a path compressing protocol the second traversal of the path will entail both the transmission of messages and the updating of the local forwarding address databases along it. If we assume that the costs of lookup and update in these databases are similar and that the results of the lookup on the first traversal are cached until the path compressing traversal, then a path compressing *access* costs about twice as much as the simpler implementations. If λ is sufficiently large compared to N for a particular application

then it may be reasonable to forego path compression.

We leave as open problems the closed form solution for the *Jacc* protocol's average case as well as the problem of deciding what kind of average case assumptions lead to improved performance. In the next chapter we argue that under the uniform choice average case assumptions that we have been making that there will not be an average case analysis that improves substantially the worst case lower bound for the *PCacc* protocol. Nevertheless, it is still an open problem to find such a bound. Finally, an area that we have not touched upon is an analysis of the effects of concurrency on the cost of executing these protocols.

Chapter 4

Numerical and Simulation Results.

In Chapter 3 we analyzed the costs of using various forwarding address protocols. The formulae with which these results are expressed are difficult to evaluate and use analytically. In this chapter we present numerical evaluations of those results in graphical form and compare them with the output of Monte Carlo simulations of the same protocols.

4.1 Simulations.

We used a Monte Carlo simulator capable of simulating each of the three protocols analyzed in Chapter 3. Each data point presented in this section is the result of running the simulator on a problem instance consisting of k operations where $k \geq max(50000, 4N \cdot max(\lambda, 1/\lambda))$, where N is the number of processors participating in the protocol and λ is the ratio of accesses to moves in the problem instance. The method by which the sequence of operations in an instance is generated is equivalent to sampling without replacement from a set of operations containing $\lambda k/(1+\lambda)$ accesses and $k/(1+\lambda)$ moves. The source processor of each access in this set is chosen from a uniform random distribution. Similarly, the destination processor of each move is also chosen uniformly. In addition to computing the mean and standard deviations of the cost of the accesses, the simulator also keeps track of the maximum cost incurred by any single access.

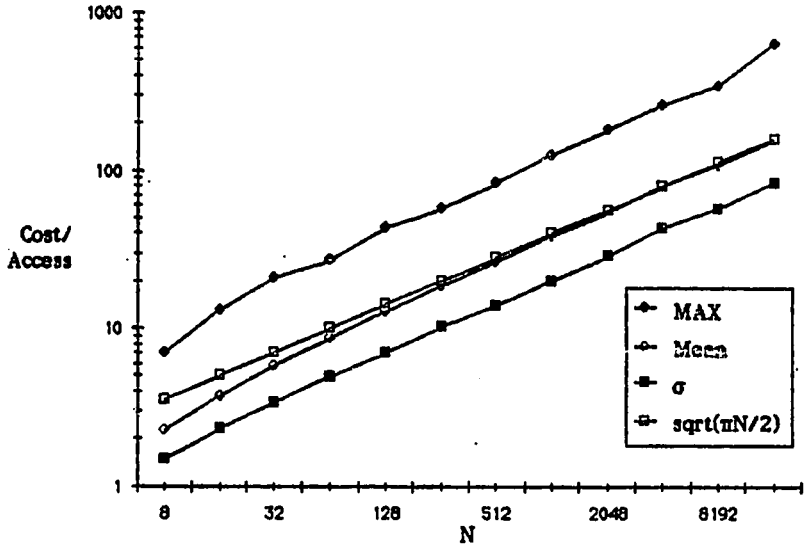

Figure 4.1: Results of simulating the *Lacc* protocol with $\lambda = 1$. "MAX" is the the maximum cost of an access. "Mean" is the mean cost of an access. "σ" is the standard deviation of the cost. Our estimation ($\sqrt{\pi N/2}$) of the mean is plotted for the purposes of comparison as is $\sqrt{2\pi N}$, the height of a random tree.

The average case problem simulated for *Jacc* is slightly different from the one analyzed in Chapter 3. In those analyses we assumed that the probability α that a particular operation will be an access is independent of all other such choices. This corresponds to a process that samples with replacement. On the other hand, the worst case bounds for both *Jacc* and *PCacc* are expressed in terms of λ, the actual ratio of accesses to moves. In order to make the results of the simulations, including those of *Jacc*, comparable with the worst case bounds we used sampling without replacement. This forces the actual ratio of accesses to moves in each simulation run to be λ exactly.

Figure 4.1 presents the results of simulating the *Lacc* protocol. The results of the simulation are consistent with the analytic bounds derived in Chapter 3. The mean cost per access converges rapidly to $\sqrt{\pi N/2}$ as N grows. Furthermore the cost of the most expensive access operation operation is bounded from below by $\sqrt{2\pi N}$, the expected height of a random rooted, labeled tree. The actual maximum cost seems to be predicted

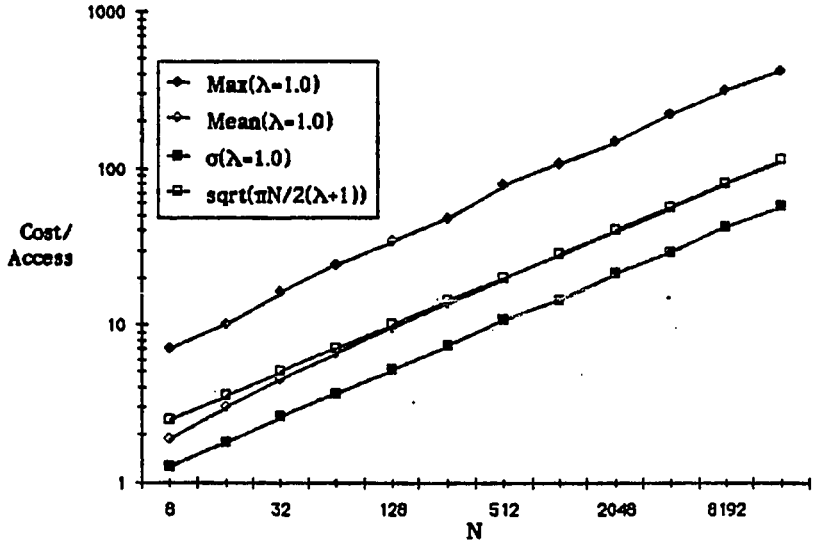

Figure 4.2: Results of simulating the *Jacc* protocol with $\lambda = 1$. "Max" is the the maximum cost of an access. "Mean" is the mean cost of an access. "σ" is the standard deviation of the cost. Our estimation $(\sqrt{\pi N/2(\lambda+1)})$ of the mean is plotted for the purposes of comparison.

well by twice this figure. When $N = 16384$ the standard deviation of the cost of an access was measured to be $\sigma = 84.6$. Since $k = 65536$ the estimation of the standard error in measuring the mean, $\sigma/\sqrt{k-1}$, is 0.33. This is the largest estimated standard error of any of the data points we present.

Figure 4.2 presents the results of simulating the *Jacc* protocol with $\lambda = 1$. Our numerically derived estimate of the mean $(\sqrt{\pi N/2(\lambda+1)})$ is also plotted for comparison. Unlike the *Lacc* protocol, an access in the *Jacc* protocol modifies the forwarding address tree and thus costs are affected by λ. In Figure 4.3 we plot the mean cost of an access for various values of N and λ. Figure 4.4 presents the maxima of the access costs encountered in those executions of the simulator.

Figures 4.5, 4.6, and 4.7 present the results of similar simulations using the *PCacc* protocol. Because the costs grow much more slowly in this case than for the other two protocols a linear rather than logarithmic scale is used for the costs in the latter two

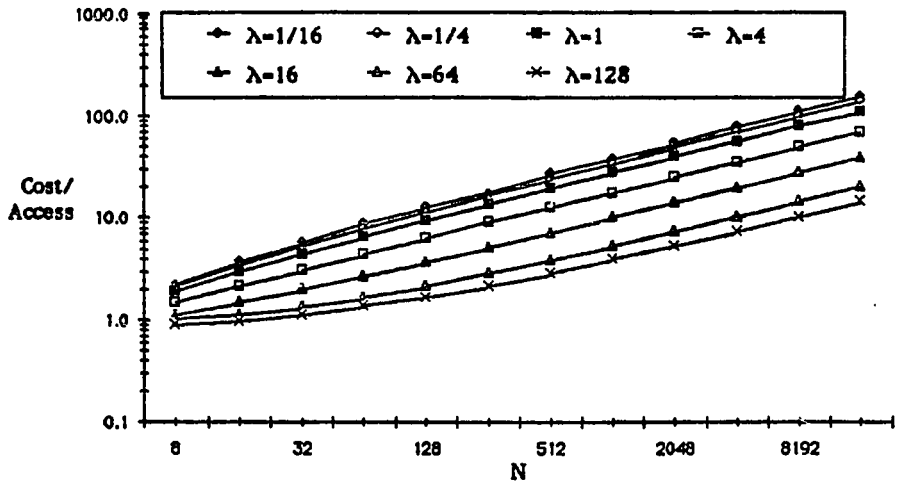

Figure 4.3: The simulated mean cost per access using *Jacc* is plotted as a function of N for various values of λ.

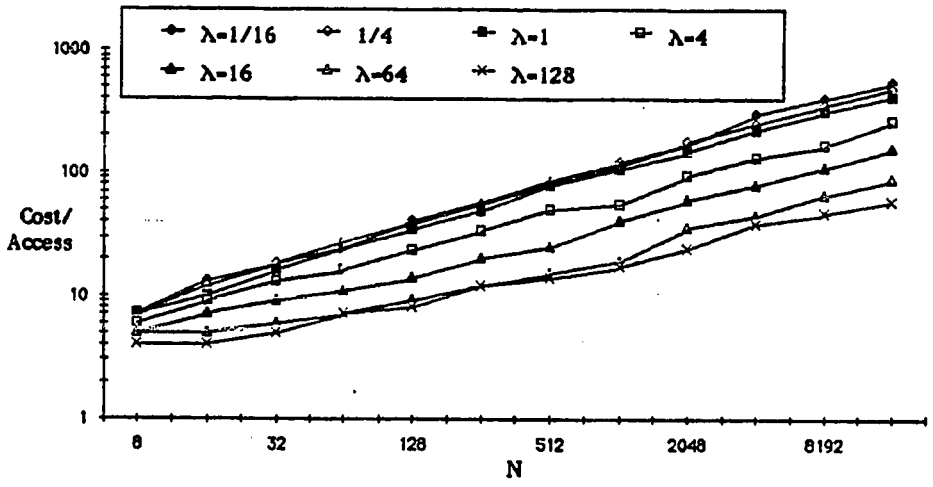

Figure 4.4: The maximum cost per access encountered in simulations using *Jacc* is plotted as a function of N for various values of λ.

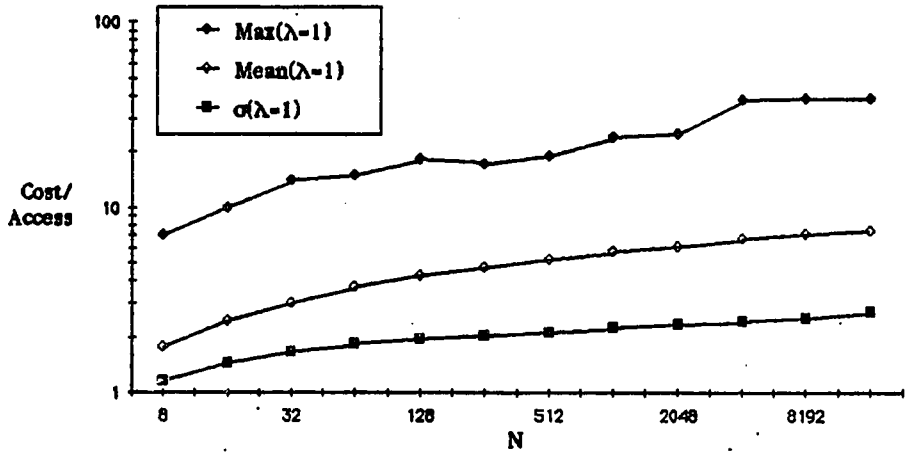

Figure 4.5: Results of simulating the *PCacc* protocol with $\lambda = 1$. "Max" is the the maximum cost of an access. "Mean" is the mean cost of an access. "σ" is the standard deviation of the cost.

graphs. In the next section we will compare these results with our analytically derived bounds on the cost of using *PCacc*.

4.2 Evaluating the bounds for *PCacc*.

In Chapter 3 we derived a worst case upper bound of the ammortized cost of the path compressing protocol. In order to compare it with the lower bound we simplified and weakened it. Here we evaluate the function numerically and compare it with the approximations, with our worst case lower bound, and with the simulation results.

The first issue is the comparison of the three bounds that we derived. The tightest bound that we derived, inequality 3.6 can be written as

$$C(N,a,m) < m + a\,(1 + \rho \ln N) + \frac{a \ln N}{\lambda \ln \frac{e^{1/\rho}}{(e^{1/\rho}-1)}},$$

where ρ is a free parameter that can be chosen to minimize the right hand side of the inequality. Henceforth we will call this *UB1*. We numerically optimized the right hand

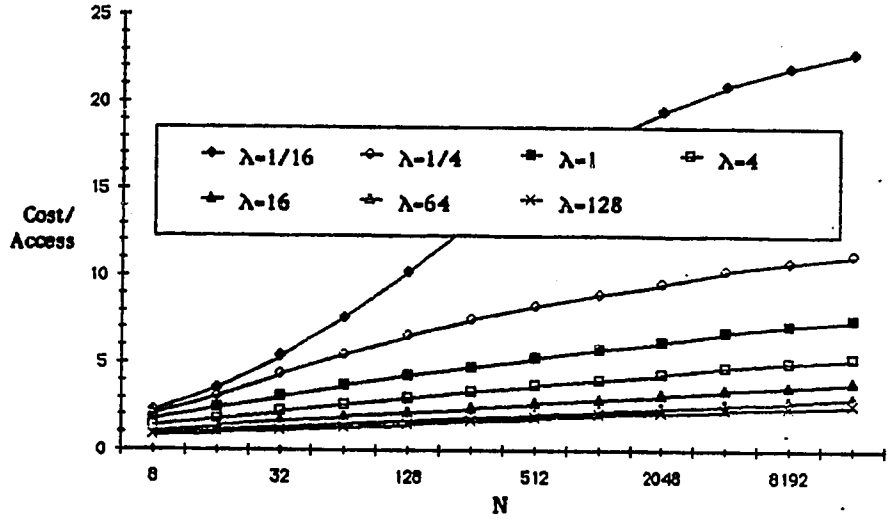

Figure 4.6: The simulated mean cost per access using *PCacc* is plotted as a function of N for various values of λ.

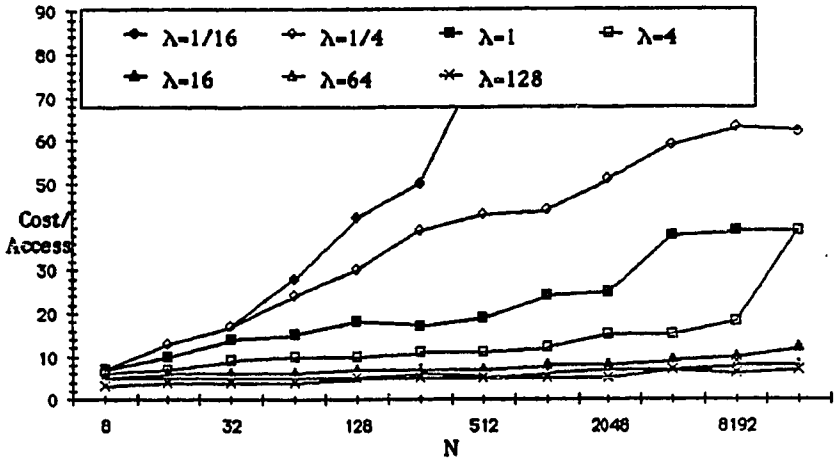

Figure 4.7: The maximum cost per access encountered in simulations using *PCacc* is plotted as a function of N for various values of λ.

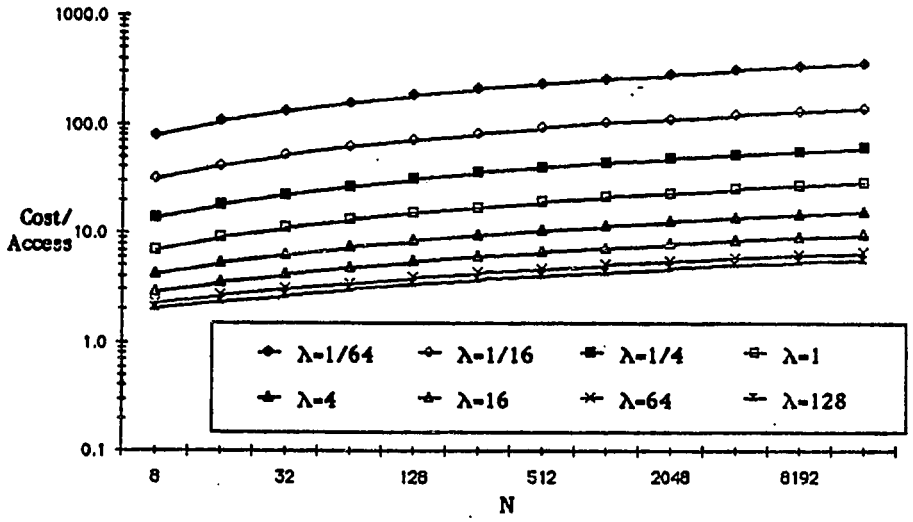

Figure 4.8: Our tightest upper bound of the worst case ammortized cost per access for the *PCacc* protocol, *UB1*, is numerically optimized and plotted as a function of N for several values of λ.

side of this inequality and the results are presented in Figure 4.8. Notice that when $\lambda < 1$ and N is small that *UB1* is worse than the trivial upper bound of $N - 1$.

A weaker but simpler bound,

$$C(N, a, m) < m + a + \left(\rho + \frac{e^{1/\rho}}{\lambda}\right) a \ln N,$$

was obtained from *UB1* where again ρ can be chosen to minimize the right side of the inequality. We call this *UB2*. Choosing ρ such that $\lambda\rho^2 = e^{1/\rho}$ optimizes *UB2*. For neither *UB1* nor *UB2* do we have an analytic solutions for choosing an optimal ρ.

Finally, by setting ρ non-optimally to to $\rho = 2/\ln(1+\lambda)$ we obtained the even weaker upper bound of

$$C(N, a, m) < m + a + (3a \log_{1+\lambda} N), \quad \lambda \geq 1.$$

We refer to this weakest bound as *UB3*.

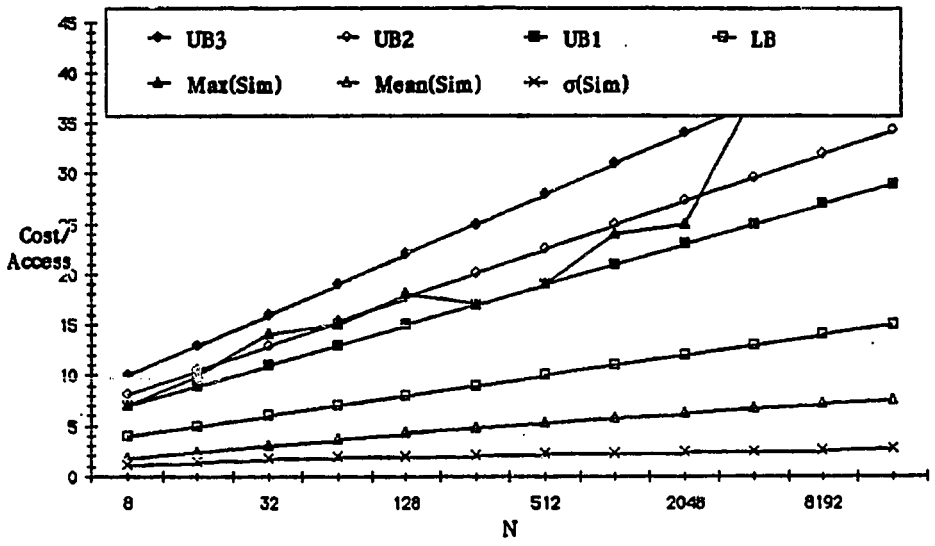

Figure 4.9: Three upper bounds, the lower bound, and simulation results for the *PCacc* protocol with $\lambda = 1$.

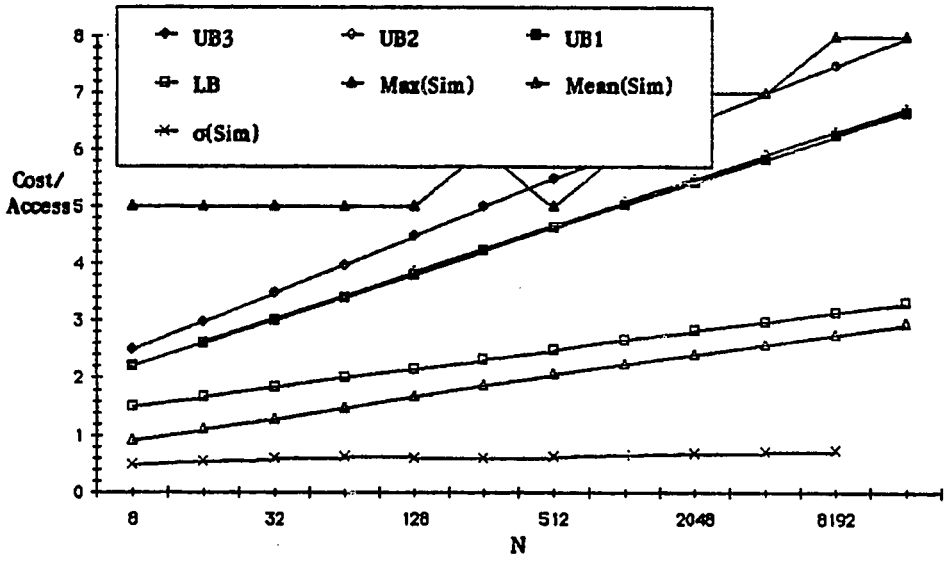

Figure 4.10: Three upper bounds, the lower bound, and simulation results for the *PCacc* protocol with $\lambda = 64$.

Figures 4.9 and 4.10 present these three bounds graphically for the particular cases in which $\lambda = 1$ and $\lambda = 64$, respectively. *UB1* and *UB2* were optimized numerically. For purposes of comparison we also plotted the corresponding lower bounds as well as the simulation results. Notice that as claimed in Chapter 3 that *UB2* is an excellent approximation to *UB1* when λ is reasonably large.

Recall that in Chapter 3 we did not present an analytic average case analysis for the *PCacc* protocol. The simulation results plotted in Figures 4.9 and 4.10 indicate that given the assumption that operations are chosen uniformly that the average case cost of an access cannot be constant but instead grows proportionally to $\log N$. When $\lambda = 1$ the mean cost per access appears to be about half of the worst case lower bound. When $\lambda = 64$ it appears to be an even larger fraction and for very large N the mean cost may approach or even exceed the lower bound. This is consistent with a simulation of the set union algorithm done by Sedgewick and reported by Yao [Yao85].

These comparisons of the results of our simulations with our analytic results lead us to believe that an average case analysis of the *PCacc* protocol that uses our uniform choice assumption will not be able to improve much upon our worst case lower bound. Average case analyses for set union algorithms have been done under certain restrictive assumptions [DR76, Yao85], but no general analysis has yet been published. As in the case of the set union algorithm the average case analysis of the *PCacc* protocol remains an open problem.

Despite the lack of a general analysis we remind the reader that there are assumptions which do imply constant worst case cost per access. If it is the case that $\lambda > \epsilon N$ for some fixed ϵ then all of the bounds for both *Jacc* and *PCacc* become constant. We leave as an open problem the investigation of other assumptions involving bounded values of λ that might lead to similar results.

Chapter 5

A Framework for Discussing Model Implementations.

In the preceding chapters we argued that, given our model of objects and naming, forwarding addresses are an obvious and natural mechanism for decentralized object finding. Given the forwarding address mechanism most applications will use it in an efficient way. In our model of costs, an object that visits N different processors over its lifetime can be accessed using the *PCacc* protocol with a worst case ammortized cost bounded by a small constant times $\log N$. We judge this to be acceptable performance.

In addition to the question of cost, there are other issues that bear upon the question of whether it is reasonable to use forwarding addresses in a real system:

- The implementation should faithfully emulate the intuitive model of objects and proper names introduced in Chapter 1. If a processor possesses a proper name then it should be able to access the referent if it still exists.
- Other resources should be used efficiently. In particular, storage should be managed in such a way that uncollectable garbage does not accumulate. While it is beyond the scope of our work to decide which objects are no longer needed, we do need to concern ourselves with recovering the storage of useless forwarding addresses.
- It should be possible to exercise reasonable administrative control over the processors in the system. Each processor should be able to account for its own resources. If a processor takes some action that commits it to future obligations then the cost of those obligations should be known.
- It should be possible to reconfigure the system. If there are obligations between processors then they should be explicitly represented by "contracts". It should be

possible to reconfigure the system by renegotiating these contracts.

- Reliability is important. There are many different sets of assumptions about failure modes in distributed systems. Multiple levels of enhanced reliability should be possible for each of these. The replication of objects and forwarding addresses should be straightforward.

These are properties that depend upon a more detailed model of the system than the simple abstraction that we used for the performance analyses. This chapter adds that detail. Aspects of system design that do not bear directly on the problem of finding objects as they move from processor to processor are actively and blatently avoided.

In this and the following chapters we explore these and other issues by presenting pseudo-code fragments that might be found in object finding services that use forwarding address mechanisms. They are not intended to convey all of the detail that would be present in actual implementations. They are instead offered as heuristic models with which to explore the ideas behind those implementations.

We remind the reader that the model we present is not the interface seen by the programmer of an application in this system, rather it is seen by the implementer of that interface. The application programmer's model might resemble Smalltalk 80 [GR83], Eden [ABLN83], Argus[Lis84], or some other "object-oriented" programming environment. The user interface might not even be recognizable as an object-oriented system.

5.1 Processor Organization.

An object location service occupies a small but fundamentally central position within the distributed system of which it is part. If we were to present pseudo-code that fully describes a location service and how it is used we would have to describe a large part of the higher levels of the system and thus appear to make architectural decisions that are not directly relevant to the problem of finding objects. We therefore present the location service as implemented on each processor as an Object Finding Package (OFP). It is a collection of procedures and code fragments that are invoked by the higher levels of the system to access the object finding mechanism. If object finding were done by a

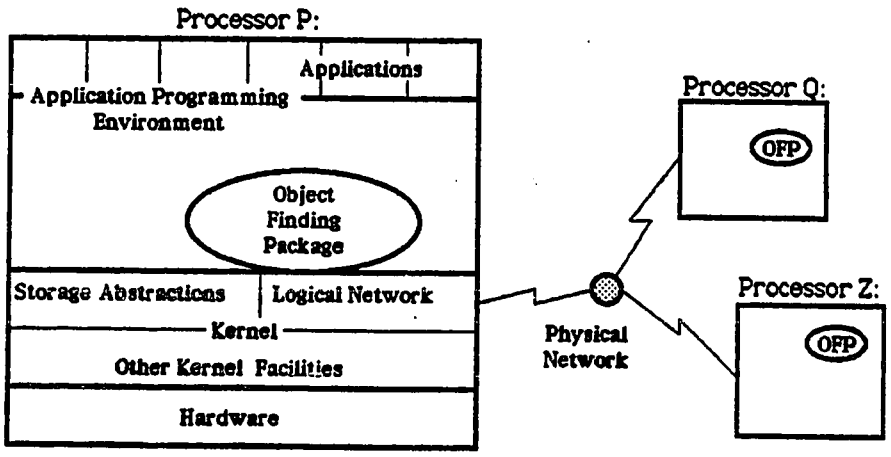

Figure 5.1: A simplified view of the software architecture of a processor in the system.

logically centralized service then the OFP would serve as the interface to that service. It is common to call such an interface an *agent*. Since the forwarding address mechanism is completely decentralized we avoid this usage.

From the point of view of an Object Finding Package, the system of the processor on which it is located is composed of three major parts. (See Figure 5.1.) First, there is the OFP itself. Second, there is a layer below the OFP that implements an underlying virtual machine that supports the language in which the OFP is implemented. This includes network communication primitives and the implementation of local permanent storage. We call this component the *kernel*. The third major component consists of everything else. One way of looking at this is that it consists of the run-time support for the language in which applications are programmed together with the applications themselves. We will lump this all together under the title of the *Application Programming Environment* or APE.

From the point of view of an application programmer the kernel, the OFP, and a large part of the APE together comprise the operating system of the processor.

The kernel provides facilities that are used both by the OFP and the APE. The APE calls the facilities of the OFP and may be required to implement procedures that are in

turn called by it.

Note once again that we are not attempting to describe a model of the system as seen by the programmers of applications, rather we are describing it from the point of view of implementing the OFP. We explicitly and actively avoid aspects of system design that do not bear directly on the problem of finding objects as they move from processor to processor.

5.2 A Pseudo-Code Language.

The language we use to present pseudo-code is loosely based on the CLU [LAB*81] programming language. We chose CLU because the language's data and control abstractions are conducive to concise presentation.

In order to suppress detail we represent elided pieces of code as comments enclosed in curly braces {*like this*}. The content of these comments are descriptions of program fragments expressed in meta–languages such as English or non–algorithmic mathematical notation. Braces are also used to denote mathematical sets and are used in the construction of CLU objects.

5.2.1 An Event Driven Control Abstraction.

The Object Finding Package is embedded in a larger Application Programming Environment. The correctness of an object finding protocol depends not only on the implementation of OFP itself but also upon it being used appropriately by higher levels of the system.

An Object Finding Package is "event driven" in the sense that certain events in the APE must cause the invocation of OFP primitives. To capture this without having to present a hypothetical implementation of the higher level that explicitly detects these events, we present the usage of the OFP primitives in a form similar to that of *guarded statements*. If *event* is a description of an event occuring on processor P and *action* is a description of an action that P can take, then the code fragment:

event $\longrightarrow$ *action*;

means that whenever an event occurs on P that satisfies the description *event* then P should execute the code *action* in reaction to that event. The details of how events are detected and how the execution of the actions are scheduled are left unspecified. We require only that the APE implementation should somehow detect all events matching our descriptions and it should react to them by executing the appropriate action.

If a variable appears in *event* and is free (not declared) in *action* then it has the same binding in both occurences.

5.2.2 An Example.

To illustrate our style of exposition we present an example, the definition of a queue of messages as might appear within the APE.

```
; CLU comments begin with a semicolon and end at
; the end of the line. These are comments.
Mqueue = cluster is create, insert, next, empty

    ; A cluster is the definition of an abstract data type.
    rep = { a pointer to an appropriate
                data structure for a queue}

    create = proc(size: int) returns (cvt)
        {Create a new queue data structure that can hold 'size' messages
        and return a pointer to it.}
        end create

    insert= proc(msg: Message, q: cvt) returns (Bool)
        {Insert 'msg' at the end of 'q' and return true if successful.
        Otherwise, there wasn't space and return false.}
        end insert

    next = proc(q: cvt) returns (Message)
        {If 'q' is not empty then remove the first element and
        return it. Otherwise return a null Message.}
        end next

    empty = proc(q: cvt) returns (Bool)
        {Return true if and only if 'q' is empty.}
        end empty
```

```
end Mqueue
```

This example illustrates the definition of an abstract data type 'Mqueue'. Such a definition is called a **cluster**. The first line lists the names of the abstract operations on an 'Mqueue'. The keyword **rep** is used as a synonym for the data structure used internally by the **cluster** to represent the abstract data type being defined. Because we are not interested in the details of the implementation we use meta–code to say that a pointer to any appropriate data structure will suffice.

The definitions of the operations defined by the abstract data type follow the definition of the representation. For instance, the operation 'create' takes a single argument, an integer specifying the maximum size of the queue to create. 'Create' builds a new data structure that can hold that number of messages and returns an encapsulated pointer to it. The keyword **cvt** is used to denote this encapsulation operation and its inverse. When an object is returned from the cluster that defines it **cvt** prevents the recipient from manipulating the underlying representation. In this case, the **return** statement in 'create' takes an object of type **rep** and converts it to an object of the abstract data type 'Mqueue'. Similarly, **cvt** in a parameter list at the interface to the **cluster** decapsulates the object to make the underlying representation accessable. For example, the procedure 'insert' is called with a 'Message' and a 'Mqueue'. The invocation protocol converts the latter into its representation.

There is an alternative infix syntax that can be used on certain common operations. If a cluster defines these operations then they can be called using an infix expression For instance the call 'Mqueue$equal(q1, q2);' can also be made using the expression 'q1 = q2'.

The message queue may be used to buffer messages as they arrive on a processor until some time later when they can be acted upon. To capture this notion we would use a pseudo-code fragment like:

```
{m : Message := Net$receive() ∧ P(m)} →
    if ¬mqueue$insert(m, q) then
```

{Handle message queue overflow.}

In this case the description of the event is intended to mean that the processor executed a statement in which the variable 'm' is declared to be a 'Message' and it is assigned as a value the 'Message' returned from an invocation of the operation 'receive' declared in the **cluster** 'Net'. Furthermore the predicate 'P' is true of 'm'. The action to be taken in response to this is is to enqueue 'm' in 'q'. Since exception handling is not the focus of the exposition, the handling of the exceptional condition of queue overflow is described informally.

5.2.3 Kernel Primitives.

The kernel component implements low level facilities that are used by the higher level components. These include:

- The kernel implements a reasonable virtual machine on top of which the rest of the system can be built in a straightforward way. We assume that it implements common facilities such as processes, virtual memory management, and inter-process communication between processes on the same processor.
- In addition, we assume that the kernel implements *stable atomic storage*. This is storage that can survive all of the common faults expected on the processor.
- The kernel incorporates a low level interface to the communication network. This includes a mechanism for uniquely identifying processors and routing messages between them.

Storage Abstractions.

In some models of processor faults it is appropriate to assume that there will be several classes of storage, each of which is affected differently by a fault. We assume that processors have only one kind of fault called a *crash*. When a crash occurs the processor performs no incorrect computations, but it does interrupt the current computation and destroys some memory. The part of the processor's storage that is erased by a crash will be called its *volatile* memory. In contrast the kernel also implements a class of *stable* storage [Gra79, Lam81] whose content is guaranteed to be reconstructible after a crash.

The existence of stable storage means that although data stored in it may be made unavailable by crash that there is a recovery procedure that guarantees that it is not really lost.

With respect to stable storage, an important concept is the notion of local *atomic actions.* They exhibit what is called *failure atomicity.* This means that if a crash occurs while the atomic action is executing then after recovery it must either appear with respect to stable storage to have completed or to have never been started. The effects on stable storage of atomic actions that have completed before the crash must not be undone by the crash. All operations on stable storage must exhibit such failure atomicity. Note that this is defined only with respect to local stable storage. A crash cannot retroactively undo any interprocessor communciation that may have been initiated while in the atomic action.

Another concept is that of an operation having *transactional atomicity.* An operation has the transactional atomicity property if no other operation can ever "see" its partially completed results. A transaction appears to be indivisible. In the presence of crashes a transaction needs to exhibit failure atomicity. In addition, it needs some form of concurrency control to protect the data it accesses from being accessed by concurrently scheduled tasks on the same processor.

We assume that the kernel implements a stable storage abstraction and that it is accessible through our exposition language by using a **stable cluster** declaration. This will guarantee that copies of all data structures declared to be own in the cluster will be kept in stable storage.

A block of pseudo–code that exhibits both kinds of atomicity can be declared by using the keyword **atomic**. This can be applied to both 'begin; end' blocks and to procedures. This denotes a mechanism for ensuring that all of the modifications to stable storage that occur syntactically within that block occur simultaneously if and only if the block is exited normally. Note that atomic actions may be invoked from within other atomic actions. These are not nested transactions in the sense used by Moss [Mos81]. As each of the nested actions is completed its effect on stable storage is made permanent

and it is not undone if a crash occurs during the execution of an enclosing atomic action.

In order to simplify the exposition we allow the pseudo-code to cause a crash by executing a **fail** statement if it encounters an exceptional condition. In a real implementation the processor would invoke an exception handler.

We assume that atomic actions exhibit appropriate mutual exclusion. The simplest method for implementing this would be to force mutual exclusion among all outermost atomic actions. Other implementations may allow concurrent execution of atomic actions that access disjoint data structures. We care only that reasonable implementations exist.

We reiterate that a local atomic action executes and accesses data solely on a single processor. Although communication primitives may appear within an atomic action they in no way implement atomic actions that involve more than one processor. Multiple-processor atomic actions must be implemented on top of these primitives.

Communication Primitives.

Another facility provided by the kernel level of the system is a simple interface to the inter-processor communication network. We assume that each of the processors on the network is given a unique name when it is connected to the system. Processor names are instances of the abstract type 'ProcID'. A processor can send a message to any other processor whose identifier it has. The routing of messages from processor to processor is handled automatically by the kernel and is hidden from higher levels of the system.

The low level interface to the network provides only an "unreliable datagram" service. Reliable message passing, remote procedure calls, and virtual circuits are not provided at this level, but may be implemented at a higher level. While we do not maintain the state necessary to maintain a true virtual circuit, we do, however, assume that the underlying network serially numbers the messages from one processor to another to ensure that all messages it delivers are also delivered in order. Messages arriving out of order are simply discarded.

The network transmits 'Messages' that are defined by

```
Message = cluster is {appropriate list of operations}

    rep = record[
              dst, src, origin: ProcID,
              type: TextString,
              body: any]

    {definitions of the operations}
    end Message.
```

A 'Message' is a record beginning with three 'ProcID' fields: 'dst', the destination for this message; 'src', the processor that is actually sending the message to 'dst'; and 'origin', the processor that originated the message and to which a reply (if any) should be directed. The message's type is represented by a text string. Finally, the type of the body of the message is unspecified. A new instance of the 'Message' data type can be created using a "constructor". For example, the constructor

```
Message${src,origin: Q, dst: P, type: "Howdy",
    body: record${x: a, y: b, z: c}}
```

creates a new message in which the 'src' and 'origin' fields are set to the value Q, 'dst' is set to P, 'type' is set to the string "Howdy", and 'body' is set to the result of constructing a record containing the three fields 'x', 'y', and 'z' that are set to the values 'a', 'b', and 'z' respectively. (This is an abuse of the notation of CLU. There should be a named declaration for type of the **record** that is assigned to 'body'.)

The network primitives are defined by the cluster Net.

```
Net = stable cluster is send, receive {, others}

    own {Data structures for routing tables, incoming message
                buffers, etc.}
    send= proc(msg: Message)
        {Send 'msg' unreliably.}
        end send

    receive= proc() returns(msg: Message)
        {Removes a Message from the input buffer if one exists.
        Otherwise, return the null Message.
```

```
        A side effect may be the updating of the routing table.}
    end receive

    {definitions of other operations and internal routines.}

    end Net
```

The assumption here is that messages are sent asynchronously and wind up in a buffer in the receiving processor. The exact details of this are not important since the OFP does not directly call *receive*. Instead it relies on parts of the APE to recognize messages meant for the OFP and for the APE to call the appropriate OFP operation.

5.3 Summary.

An decentralized object finding service that uses a forwarding address mechanism is represented on each processor in the system by an Object Finding Package or OFP. The OFP is an abstract data type that implements that mechanism. It appears to the Application Programming Environment (APE) to be the data structures for keeping track of all forwarding addresses that are "interesting" to that processor together with the operations necessary to manipulate it.

In order to facilitate the description of an OFP we assume that there is an underlying system that implements stable storage and network communication primitives.

Chapter 6

A Location Service for Atomic Objects.

In this chapter we present a simple model implementation of a forwarding address protocol for atomic objects as defined in Chapter 1. While most of exposition is the routine presentation of the pseudo–code that one might expect *a priori* to find in such an implementation, there are several points of particular interest.

- The implementation does not assume that multi–processor transactions are available as primitive operations. Modifications to the forwarding address tree of each object are instead performed as local atomic actions. The code that implements these operations is concise.
- The identity of a remotely–accessible object is defined by its interaction with the system of proper names and forwarding addresses. A particularly revealing aspect of this is the way that various concrete data structures are included or excluded from an object's representation as the protocol to move it is executed.
- The protocol for moving an atomic object implements a two processor transaction in which the "commit record" for the transaction is embedded in the forwarding address graph. This ensures that although the possibility of lost messages means that we cannot guarantee that the transaction will be committed or aborted in any fixed time, we can guarantee that it will be committed or aborted before it is next accessed.

Refinements of the model implementation dealing with improved storage management and fault tolerance through replication are dealt with in later chapters.

The pseudo–code of our model implementation differs from a real implementation in two major ways. The code necessary for handling exceptions and for performing

consistency checks would be a major part of a real implementation. For the purposes of exposition we have ignored most of this with the intention of making the essence of the protocols more intuitively accessible to the reader. The penalty for using this strategy is that rigorous arguments about the correctness of the protocols, especially in the presence of lost messages and processor crashes, are not possible because the protocols themselves are not strictly correct without all of the appropriate exception and consistency handlers.

Another major difference between our pseudo–code and a real implementation is the issue of efficiency. Our model implementation is intended to be compact and relatively straight–forward. A real implementation would entail considerable refinement in order to achieve efficiency. In particular, there would be considerable use of unstable caches that mirror the information in stable data structures. We have ignored this issue.

6.1 Data Structures.

We begin by describing the data structures used by our model Object Finding Package for atomic objects.

6.1.1 Time Stamps.

The data structures that refer to object locations will contain fields of type TimeStamp. The requirement of this implementation is that the TimeStamps associated with a particular object increase monotonically with physical time. Since new forwarding addresses need only be created when we attempt to move an object, its TimeStamp need only be updated at that time. This, together with the fact that TimeStamps of different objects are never compared, means that it is sufficient to let each object's TimeStamp be a count of the number of times that we have attempted to move it. In our exposition TimeStamps are implemented by positive integers.

An alternative to the simple counter scheme is to derive the TimeStamps from a global clock that approximates physical time. The advantage of this is that processors can use TimeStamp values to identify data that has not been updated in a long time. If this is done then they must be globally consistent with the causal ordering of events

in the system [Lam78b]. Independent local clocks are not sufficient because we must be assured that the TimeStamps associated with each object increase monotonically. The disadvantage of this strategy is that it requires a larger amount of storage.

6.1.2 Canonical Identifiers.

In presenting our model of causally connected proper names we argued that any processor that has a proper name for an object could as well have a Canonical Identifier (CID) for it. To ensure the competence of all processors to decide whether two names refer to a single object we mandated that all proper names that are transmitted between processors contain CIDs. The APE is free to hide existence of CIDs from applications programmers. It is also free to define other systems of naming and description. We only require that proper names contain CIDs and that all interaction with an OFP be in terms of CIDs. It is not necessary that all objects be baptised with CIDs, merely that all objects known to an OFP be so baptised.

Each processor needs access to a source of Canonical Identifiers. In distributed computer systems a common method for a processor to locally generate a sequence of unique identifiers is for it to concatenate its own unique processor identifier with elements from a sequence of integers that increases monotonically with real time such as values taken from a reliable local clock or counter.

The network interface of each processor guarantees the existence of unique processor identifiers. The existence of stable storage guarantees the ability to generate a monotone sequence of integers irrespective of the existence of a monotone local clock. Although an implementation may not use these, their existence does mean that it is possible to generate CIDs. The CID abstraction can be defined as follows.

```
CID = cluster is create, equal, less, copy

    rep = {an appropriate representation}

    create = proc() returns(cvt)
        return({create a new CID and return it.})
        end create
```

```
equal = proc(a,b: cvt) returns(Boolean)
        return(a = b)
        end equal

less = proc(a,b: cvt) returns(Boolean)
        return(a < b)
        end less

copy = proc(s: cvt) returns(cvt)
        return(rep$copy(s))
        end create

end CID
```

In order to use CIDs as keys for efficiently searching data structures the abstraction defines an ordering for them based on an ordering relation for the underlying representation. In most object oriented systems it is necessary to protect unique identifiers from unauthorized manipulation by the user. Since the OFP is insulated from the user by the APE we do not worry about these protection issues.

Although a CID may contain within it some location information, in particular the identifier of the processor at which an object was baptised, we do not provide an operation to examine this field. The forwarding address entries for the objects will contain more recent information as to the object's location.

6.1.3 Local Object Maps.

The Object Finding Package on processor P uses two data structures to keep track of objects that can be referenced using CID's. The first of these is a stable table called a Local Object Map or LMAP. Processor P uses its LMAP to translate the CID of an object located at P into a handle so that P can ostentively manipulate its representation. The LMAP thus contains an entry for each object located at P. In addition, it contains entries for objects that are in the process of moving to or from P.

An LMAP entry is defined as

```
LMAPentry = record[
            name: CID,
            ts: TimeStamp,
            handle: {pointer to the representation of the object},
            reallyHere, inMotion: Boolean].
```

The 'TimeStamp' field 'ts' is used to determine when the object's representation was (or will be) moved to P. The Boolean field 'reallyHere' is used to distinguish the case in which the object is really located at P from the case in which the data structure is a copy of the representation of an object actually located elsewhere. The Boolean field "inMotion" is used to mark those objects that are at P, but which are in the process of being moved elsewhere. It serves as a lock to freeze the representation of the object while it is in motion.

The LMAP itself is defined as

```
LMAP = stable cluster is create,lookup,insert,delete,update,elements

rep = { a table of LMAPentry }

own lmap: rep

create = proc
            { Create and initialize the table 'lmap'. }
        end create

lookup = proc(name: CID) returns(LMAPentry)
            { Use 'name' as a key to search the LMAP for a matching
            entry and return a copy of it.
            If no entry exists then return NULL.}
        end lookup

insert = proc(ent: LMAPentry)
            { If an entry with key 'ent.name' does not exist then
            insert 'ent' in the table.
            Otherwise fail.}
        end insert

delete = proc(name: CID)
            { If an entry with key 'name' exists then delete it.
```

```
            Otherwise fail.}
        end delete

    update = atomic proc(ent: LMAPentry))
            { If an entry with key 'ent.name' exists then
            replace it with 'ent'.
            Otherwise fail.}
        end update

    elements = iter () yields(LMAPentry)
            j: LMAPentry;
            for j ∈ lmap do
                yield(j);
            end
        end elements

    end LMAP.
```

The entry for an object 'handle' in an LMAP is as close as we will get to the actual representations of objects. Note that the definition of the LMAP contains an iterator called 'elements' that allows the APE to enumerate its members. While we do not use 'elements' in our presentation it is included because it is useful for housekeeping operations.

6.1.4 Forwarding Addresses and Forwarding Address Tables.

In this model implementation each processor keeps a single forwarding address for each atomic object it can reference. Each forwarding address points to a single processor. Forwarding addresses are represented by data structures defined as follows.

```
FA = struct[
            name: CID,
            loc: ProcID,
            ts: TimeStamp].
```

In CLU a 'struct' is an immutable entity. A forwarding address can be copied, but once created its components cannot be modified.

The forwarding addresses at each processor are kept in a Forwarding Address Table (FAT). A FAT is used to map from CIDs to forwarding addresses for the objects they name. The Forwarding Address Table on processor P contains an entry for each CID that can be used by applications running on P. In addition, it also required to maintain other entries in order to help other processors. By definition the only way that a forwarding address pointing to a particular processor can exist is if the named object has been located at that processor. P is therefore also required to keep a forwarding address for any object x with these three properties:

1. x has been located at P.
2. Another processor has had a forwarding address for x that points to P.
3. x might still exist.

The entries in a Forwarding Address Table are forwarding addresses augmented with a Boolean field called "seenHere". This field is set to **true** on processor P if and only if the corresponding object has been located at P and another processor can have a forwarding address that points to P.

```
FATEntry = record[
            fa: FA,
            seenHere: Boolean]
```

The Forwarding Address Table itself is defined as:

```
FAT = stable cluster is create, add, delete, lookup, update, elements

    rep = { an appropriate concrete representation }

    own fat: rep

    create = proc()
        {Create and initialize the table 'fat'.}
        end create

    delete = proc(f: FA)
```

```
        {Delete 'f' from the FAT.
        Otherwise fail.}
        end delete

    lookup = proc(id: CID) returns(FA)
        {Return the forwarding address for 'id' if it is in the FAT.
        Otherwise returns the NULL forwarding address.}
        end lookup

    update = proc(f: FA, always, setSeenHere: Boolean) returns(Boolean)
        { Look up the entry for 'f.name'. (Call it 'ent').
        If it exists then the timestamps are compared and
        if 'f' is newer than 'ent' then replace it with 'f'.
        If 'always' and no entry exists then create one using 'f'.
        If 'setSeenHere' then set 'seenHere' in the entry to true.
        If the table was modified then return true.
        Otherwise return false.}
        end update

    elements = iter () yields(FATEntry)
                j: FATEntry;
                for j ∈ fat do
                    yield(j);
                end
        end elements

    end FAT
```

The representation of the table should be chosen with an eye towards efficiency. Because reading from stable secondary storage is comparatively expensive the underlying implementation of the FAT should attempt to minimize disk accesses by using an appropriate caching strategy.

6.2 Local Manipulations of Objects

The Object Finding Package is not directly concerned with the semantics of accessing and manipulating objects as they appear to the programmer of an application. This is the domain of the Application Programming Environment.

The OFP becomes involved only when the APE is using proper names containing CID's to reference objects, or when CID's or objects are being moved or copied between processors.

6.2.1 Local Access to Objects

A processor can ostentively manipulate any local object. If an object will only be referenced locally then there is no need to give it a CID or to inform the location service of its existence. The programming system can use a "compiler style" rather than an "operating system style" implementation for these objects. There are considerable performance advantages to this approach.

If processor P has a CID x it can use its LMAP to determine whether the referent of x is local and if so, then it can get an ostentive reference to it.

Remote access of an object named by a CID involves sending a request message to the processor that has its representation. The semantics of the manipulation are in the province of the APE. The OFP's responsibility for the access ends when the request message is delivered to the correct processor.

6.2.2 Baptising Objects.

We will call the act of binding a CID to an object its "baptism". This has the effect of registering it with the OFP. Before being baptised an object will have been created and then perhaps manipulated ostentively at the processor P at which it was created. To baptise it the APE on P calls the procedure 'baptise' with a handle for the object.

```
baptise = atomic proc(obj: OHandle) returns(CID)
    c: CID := CID$create() ; Get a new CID for the object.
    t: TimeStamp := 0 ; Get a zero TimeStamp.
    if LMAP$insert(LMAPentry${name: c, ts:t, handle: obj,
                    reallyHere: true, inMotion: false})
    then if FAT$update( FA${name:c, loc: P, ts: t}, true, true)
        then return(c)
        else fail
        end
    end baptise
```

In this fragment a CLU "constructor" is used to build an LMAPentry that is inserted into the LMAP. Note that this version of 'baptise' is conservative in that it sets 'seenHere' as soon as the object is baptised under the assumption that the CID will eventually be exported. Other implementations might actually wait until then to set this flag.

6.2.3 Deleting an Object

Baptism is the act of creating LMAP and FAT table entries for an object. Destroying those entries at the object's location is the inverse operation. The object's representation may continue to exist but it will never again be accessable through its former CID. While the CID is now a dangling reference it will never be usable in the future to refer to a different object.

Our assumption that Forwarding Address Tables and Local Maps are kept in stable storage unaffected by processor crashes means that if an attempted access following a chain of forwarding addresses ends with a processor that replies "Unknown Object" then the object itself was deleted. A processor that receives this reply can delete its forwarding address for the object. It can also inform other processors of the fact that the object does not exist.

The decision to delete an object can be made only by the processor at which it is located. We do not specify the rules for deletion rather we leave that for the definition of the user interface defined by the APE. Object deletion could be done as a result of garbage collection or as a result of an explicit deletion operation.

{It is consistent with the APE's semantics to delete
the referent of CID x} $\longrightarrow$

objectDelete(x)

The operation 'objectDelete' could be implemented as follows.

```
objectDelete = atomic proc(name: CID) returns(Boolean)
        lme: LMAPentry:= LMAP$lookup(name)
        if lme.reallyHere then ; Only works if object is local.
            LMAP$delete(name)
```

```
                FAT$delete(name)
                return(true)
            else return(false) end
        end objectDelete
```

6.3 Moving an Atomic Object.

When an atomic object is stationary its identity is defined and protected by the processor at which it is located. The APE at that processor is responsible for this. It can take any actions consistent with the data abstraction aspects of the object and with the requirement that the object be analagous to a single physical entity with a unique identity. It can relocate the object locally, change the implementation of its representation, and manipulate it in any appropriate manner. All that we require is that if the object has been baptised that the 'handle' stored in the processor's LMAP be a valid local (ostentive) reference to it.

When an object is moved between processors its identity must be preserved. In particular, the move operation must ensure that all proper names that refer to the object continue to do so. An encoding of the representation of the object must be transferred between the processors and the system of proper names be manipulated in such a way as to guarantee that the object is neither lost nor inadvertently replicated.

In order to meet these conditions the moving of an atomic object must be a multi-processor atomic operation, or *transaction* involving both the old and new locations. There are two atomicity properties that are relevant. *Failure* atomicity means that when the system recovers from a failure that occurs during a move then it must appear either that the move was completed or that it was not started. A failure cannot result in the loss of the object, the creation of a clone, or having the object appear to be in some intermediate state. *Synchronization* atomicity means that while the move operation is in progress that intermediate states of the object cannot be seen by a concurrently executing operation.

6.3.1 Using Forwarding Addresses as Commit Records.

A standard method for implementing multi–processor transactions in a system in which individual processors are capable of local atomic actions is the "two–phase commit" protocol [Gra79, Lam81]. In the first phase of the protocol all of the participating processors attempt to take local atomic actions that record the results of the transaction if it should be completed successfully, or committed. If a processor succeeds in the first phase then it reports this fact to a processor that serves as the *transaction coordinator.* Eventually the coordinator completes the transaction in a single local atomic action by writing what is known as a *commit record* for the transaction. If all processors reported success in the first phase then the commit record can record that the transaction was successfully *committed.* Otherwise it must record that it was *aborted.* Eventually all processors participating in the transaction learn the content of the commit record, even in the presence of failures. Only when it learns that the transaction has committed can a processor make the results permanent and visible.

We present a protocol in which the forwarding address mechanism is used easily and naturally to implement a two processor transaction for moving an atomic object. Recall from Chapter 1 that if changing a data item affects the abstract state of an object then it must be considered to be part of the object's representation. If an object x is located at processor P then the FAT and LMAP entries at P must be consistent with x being located there. Because of this property these two data items must be regarded as part of x's representation. We use this to commit the move. In phase one of a protocol to move x from P to Q the representation is frozen at P and a copy is sent to Q. Q stores the copy in stable storage and reports the success of phase one to P. If phase one is successful then phase two consists of changing the LMAP and FAT entries to be consistent with a successful move. The move is committed when the FAT and LMAP entries at P are changed to be consistent with the move of x to Q. In taking this local atomic action these entries cease to be part of x's representation. P can abort the transaction by advancing the timestamp in its FAT entry for x. At that point it is free to unfreeze its copy of the representation.

At this point in the protocol Q has a stable copy of the representation, but does not yet know whether the object has really moved. It will discover this fact whenever it receives a message containing a new forwarding address for x claiming that x is at Q and is thus consistent with the success of the move. When this occurs Q can finally create FAT and LMAP entries pointing to itself. At this time these entries become part of x's representation. Q will learn that the move failed when it receives a message containing a forwarding address for x that is inconsistent with the success of the move. This information can find its way to Q in a variety of ways. P may send a "Bulletin" message to Q immediately after a move is committed. Q may query P using a "Whereis" message if the move is not completed in a reasonable time. Finally, some processor, including P and Q, may wish to access x. Until Q has updated its FAT entry for x all forwarding address paths for x will go through P.

6.3.2 The Move Protocol in Detail

The key to the move protocol is the interpretation of FAT and LMAP entries. A named object x is really at processor P if and only if the forwarding address in the FAT entry for x at P points to P. Furtheremore, there must be an LMAP entry for x at P. Since there will also be LMAP entries used to keep track of copies of the representations of objects being moved to P, the Boolean field 'reallyHere' in the LMAP entry is used to distinguish these two cases. If the object is in the process of being moved *to* P then the FAT points to the old location and 'reallyHere' in the LMAP entry is set to false. P can thus scan its LMAP to discover object moving transactions that have not completed either by commiting or aborting. When an object is being moved *from* P the Boolean field 'inMotion' in the corresponding LMAP entry is set to true. This locks the local copy of the representation until the transaction is either commited or aborted. Breaking the lock causes an abort.

Because messages can be delayed, it is necessary that each attempted move of an object be distinguishable from all others. We enforce this by advancing the TimeStamp in the FATEntry at the object's old location every time there is an attempt to move it.

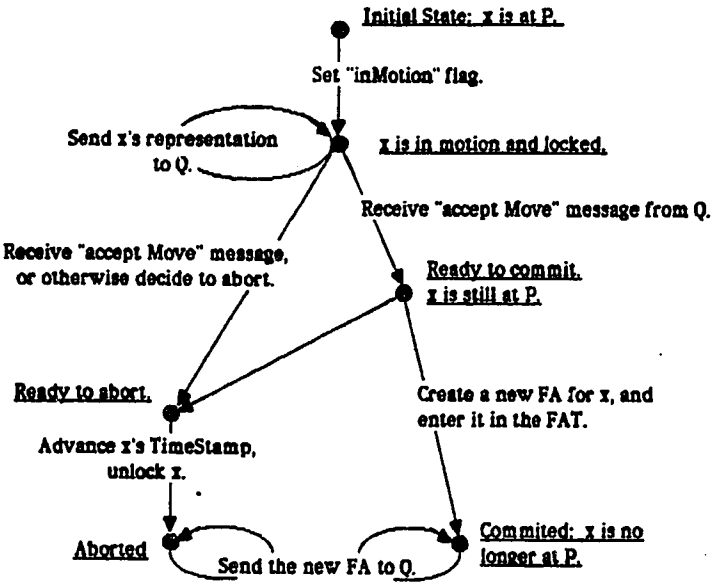

Figure 6.1: Moving x from P to Q. A finite state diagram for the transitions at P.

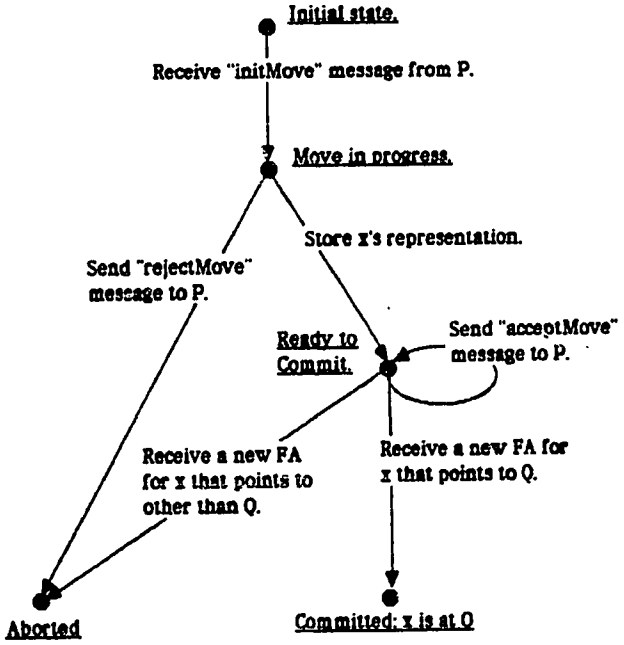

Figure 6.2: Moving x from P to Q. A finite state diagram for the transitions at Q.

The possible changes to these data structures at P, the old location of x, during the course of an attempt to move it to Q are represented in Figure 6.1 as a finite state transition diagram. The corresponding changes at Q are represented in Figure 6.2. This method of representing the transitions in the transaction protocol is well known [SS81]. Any code that implements these transitions as local atomic actions can serve as an acceptable implementation of the the protocol. In our model implementation much of it is distributed through the protocols for accessing objects and for locally maintaining the FAT and LMAP tables.

To initiate an attempt to move an object a processor can execute the procedure 'initMove'.

{Processor P wants to move the referent of x to Q} $\longrightarrow$

```
initMove(x, Q)
```

```
initMove = proc(x: CID, Q: ProcID)
    lme: LMAPentry := LMAP$lookup(x)
    atomic begin
        if (¬ lme.reallyHere ∨ lme.inMotion) then
            { "Inappropriate to initiate a move" exception.}
        else
            lme.inMotion := true
            LMAP$update(lme) ; x's representation is now locked.
            now: TimeStamp := FAT$lookup(x).fa.ts
            end
        end ; of the atomic action
        while now = FAT$lookup(x).fa.ts do
            ; Retry periodically until the transaction terminates.
            Net$send(Message${src, origin: P, dst: Q, type: "initMove",
                body: record${f: FAT$lookup(x),
                        rep: {x's representation}}})
            delay({an appropriate timeout period})
            end
        end initMove
```

The procedure 'initMove' begins by executing the first transition of the move protocol as a local atomic action. This locks x's representation and locally identifies the transaction with the TimeStamp 'now'. It then periodically tries to send an "initMove"

message to Q as long as 'now' is the current TimeStamp. When it is no longer current transaction has terminated, one way or another. Note that we assume that the other actions associated with the move can be executed concurrently with the while loop of 'initMove'.

Processor Q reacts to the receipt of an "initMove" message in the following way.

```
{ m := Net$receive() & m.type = "initMove" } ⟶

    if { Q does not want to accept x. } then
        Net$send(Message${src,origin: Q, dst: m.origin, type: "rejectMove",
            body: record${f: m.body.f}})
    else atomic begin
        x: CID := m.body.f.name
        FAT$update(m.body.f, false, false) ; Point to where x is coming from.
        h : handle := {Copy m.body.rep to stable storage and return a handle.}
        now : TimeStamp := m.body.f.ts
        LMAP$insert(LMAPentry${name: m.body.f.name, ts: now, handle: h,
                reallyHere, inMotion: false})
        end ; of the atomic action
        ; We assume that the atomic action was successful
        while FAT$lookup(x).ts = now do
            Net$send(Message${src,origin: Q, dst: m.origin, type: "acceptMove",
                    body: record${f: m.body.f}})
            delay({an appropriate retry period})
            end
        end
```

When P, the originator of the move receives an "acceptMove" message it can execute the following fragment to commit the move.

```
{ m:=Net$receive() & m.type = "acceptMove"}⟶

    atomic begin
        tfa: FA := m.body.f ; Copy it into a temporary.
        lme: LMAPentry := LMAP$lookup(tfa.name)
        if lme ≠ NULL & lme.inMotion & lme.ts = tfa.ts then
            ; This transaction is still active. Commit it.
            newfa: FA := FA${name: tfa.name, loc: m.origin, ts: tfa.ts + 1}
            FAT$update(newfa)
```

```
        LMAP$delete(tfa.name)
        end
    end ; of the atomic action
; No matter how the transaction turned out send Q an FA for x.
Net$send(Message${src,origin: P, dst: m.origin, type: "Bulletin",
        body: FAT$lookup(tfa.name)})
```

Processor P uses its LMAP entry for x to decide whether it is attempting to move it and whether this "acceptMove" message is part of the currently active attempt. If it is, then the transaction is committed. In any case, Q is informed of x's current status.

Until the transaction is commited P may decide to abort it. This may occur for several reasons. P may receive a "rejectMove" message. There might be a timeout for the transaction. P might receive a request to access the object and decide to service that request rather than complete the move. In all of these cases, P executes the following fragment to abort the move.

{P decides to abort the move of object x.}⟶

```
atomic begin
    f: FA := FAT$lookup(x)
    lme: LMAPEntry := LMAP$lookup(x)
    newtime: TimeStamp := f.ts + 1
    lme.inMotion := false
    f.ts, lme.ts := newtime
    LMAP$update(lme)
    FAT$update(f)
    end ; of the atomic action
```

These code fragments implement all of the transitions in the transition diagram of the two–phase commit protocol for moving an atomic object x except for the transitions in which the new location of the object, processor Q, learns whether or not the transaction has been committed. These transitions are triggered by the arrival at Q of a forwarding address for x with an appropriately new TimeStamp. If the new forwarding address names Q as x's location then the transaction was committed, otherwise it was aborted.

The logic for these transitions is embedded in the code of 'registerFA', a procedure that is invoked for all newly arrived forwarding addresses.

```
registerFA = atomic proc(f: FA, always: Boolean)
    lme: LMAPentry := LMAP$lookup(f.name)
    if lme≠NULL & f.ts≥lme.ts& ¬lme.reallyHere then
        ; A move to Q is in progress and 'f' can complete it.
        if f.loc = Q then ; Q is the ID of this processor.
            ; Commit the move.
            lme.reallyHere := true
            lme.ts := f.ts
            LMAP$update(lme)
            FAT$update(f, true, true)
        else ; Abort the move
            LMAP$delete(f.name)
            FAT$update(f, always, false)
             end
        elseif lme = NULL then
            ; an ordinary update of the FAT FAT$update(f, always, false)
        else
            {Q has an LMAP entry for 'f.name',
            but 'f' does not complete a move.
            'f' might be out of date or there might be inconsistencies.
            Handle it as an exception.}
        end registerFA
```

Whenever a forwarding address is received at Q it should execute the 'registerFA' procedure. If the argument 'always' is **true** then a new 'FATEntry' will be created if none exists. Otherwise 'registerFA' will only update the FAT as appropriate. It returns **true** if and only if the FAT was modified. We will illustrate the actual usage in the following sections. As we noted above, our pseudo–code is organized for the purposes of exposition rather than efficiency. A real implementation would probably integrate the functionality of 'registerFA' with the implementations of the FAT and the LMAP.

6.3.3 Discussion.

A decision that must be made in the design of any transaction protocol using two–phase commit is how perform recoveries from lost messages and processor crashes. That is,

after a crash the protocol should designate the processor or processors responsible for initiating the actions that will cause a pending transaction to commit or abort and cause all participating processors to learn the outcome. By embedding the protocol to move an object at a low level within the mechanism that defines the binding between proper names and the data structures representing objects we have a simple and natural resolution to this issue. Any subsequent attempt to refer to the object being moved must drive the transaction to completion if it is to succeed. Furthermore, any attempt to refer to the object at the new location with a forwarding address containing a new timestamp conveys the information that the transaction has completed. Although lost messages and crashed processors may indefinitely delay the completion of the move, it is thus forced to be correctly serialized with attempts to access it.

If the object being moved is an active object capable of spontaneously waking up then it may be that it will never be accessed by other objects. This can be handled by stably scheduling a wakeup operation as part of the protocol for transmitting the object's representation.

Hewitt's "Actors" formalism [HBS73, HB77] is an object oriented model of distributed computation in which spatially separated objects called *actors* can communicate and interact only by moving intermediary actors between them. This is the basic computational step out of which more complex actions are built. In a similar vein our object moving transaction could be used as a building block for more complex reliable distributed computations. We leave the investigation of the practicality of this for the future.

6.4 Managing CID's.

A processor must have a valid forwarding address for a CID in order for it to be useful. If an application causes a CID to be sent between a pair of processors then it must be accompanied by a forwarding address. This can be appended to the message containing the CID.

When an application receives an appended forwarding address from another processor, it is the responsibility of the APE to use the 'registerFA' operation defined above to register the new information in the forwarding address table.

```
{CID 'x' appears in an outgoing message 'm'.} ⟶
        {Append 'FAT$lookup(x)' to the end of 'm'.}

{Forwarding Address 'f' arrives appended to message 'm'} ⟶
        registerFA(f, always: true)
        {Strip 'f' from 'm'.}
```

Note that the parameter 'always' to 'registerFA' is set to true to create a FAT entry if one did not previously exist.

Our exposition of the model implementation does not specify how the APE manages the storage of CID's. We do, however, require that it implement a procedure called 'possiblyKnownLocally' that the OFP can call to decide whether it might be able to discard useless forwarding addresses.

```
possiblyKnownLocally = atomic proc(c: CID)
    if { No local application has access to a copy of 'c'.}
        then return(false) end
        else return(true) end
    end possiblyKnownLocally
```

The APE should use a strategy for storing CID's so that 'possiblyKnownLocally' can be evaluated effectively. Candidate strategies include tagged memory architectures and the use of C-lists [Lev84]. If applications are allowed to hide CID's within their data structures in a way that the APE cannot be guaranteed to recognize them then it must always return true from 'possiblyKnownLocally'.

Given an implementation of 'possiblyKnownLocally' that can correctly return false a processor can delete truely useless FAT entries. A maintenance process may want to execute the following loop.

```
for fae: FATEntry in FAT$elements() do
```

```
        atomic begin
            if ¬ possiblyKnownLocally(fae.fa.name) & ¬ fae.seenHere then
                FAT$delete(fae.fa)
            end
        end ; of the atomic action
    end
```

6.5 OFP–Specific Communication.

While most of the communication in the system is initiated by applications, an APE and its OFP can send information and requests to the OFP's on other processors.

6.5.1 The "Bulletin" and "Where is" Messages.

At any time a processor P is allowed to send a "Bulletin" message to another processor. This "Bulletin" message contains copies of the forwarding addresses from P's Forwarding Address Table for some set of objects. It is intended for use only by the OFP and the APE of the receiving processor should pass it on without examining its contents. It is identified by the string "Bulletin" in the message type field and its body is a forwarding address.

The receiver of a "Bulletin" message updates existing FAT entries but it does not create any new ones. It does this by calling 'registerFA' with the paramater 'always' set to false. As discussed above this may have the effect of completing a move transaction.

A processor may send a "WhereIs" message to another in order to provoke the return of a "Bulletin" message for some object. The body of the message consists a CID. The expected reply is a "Bulletin" message containing a forwarding address for that CID request. If the replying processor does not have a FAT entry for it then it sends back a forwarding address with a TimeStamp of 0 that names the NULL processor ID as the object's location.

A processor P wanting to send one of these messages to Q constructs it as follows.

```
bulMess: Message := Message${src,origin: P, dst:Q, type: "Bulletin",
                             body: {a FA goes here}}
```

whMess: Message := Message${src,origin: P, dst:Q, type: "WhereIs",
body: {a CID goes here.}}

6.5.2 Path Compression.

Assume that processor P has a forwarding address for object x that points to processor Q. If P learns that x was subsequently located at Q' then it can initiate one stage of path compression by sending a message of type "PathUpdate" to Q. The informally stated meaning of a "PathUpdate" message is: "I, processor P, used to have a forwarding address for x that pointed to you, processor Q. I will no longer use that forwarding address because I have received f, a more recent forwarding address, that points to Q'." A "PathUpdate" message is constructed as follows:

puMess: Message := Message${src,origin: P, dst:Q, type: "PathUpdate",
body: {A FA goes here.}}.

"Bulletin" and "PathUpdate" messages contain similar information. They are given distinct types because the former can contain forwarding addresses that are useless to the destination processor. The content of a "Bulletin" is strictly advisory information. In contrast, a "PathUpdate" message conveys the added information that one stage in a path compression has been performed and serves specifically as an invitation to continue the path compression if possible.

A processor P reacts to a "PathUpdate" message as follows:

```
{m := Net$receive() & m.type="PathUpdate"} ⟶
    g := FAT$lookup(m.body)
    if m.body.loc = NULL then ; Propagate "object deleted"
        FAT$delete(f)
        Net$send(Message${src, origin: P, dst: g.loc, type: "PathUpdate",
            body: m.body})
    elseif FAT$update(m.body, false) then
        ; The update was successful because 'm.body' is newer than 'g'.
        ; Propagate the path compression.
        Net$send(Message${src, origin: P, dst: g.loc, type: "PathUpdate",
            body: m.body})
        end
```

If the "PathUpdate" messages contains a NULL forwarding address indicating that the object has been deleted then it is propagated as far as it can be. Otherwise the path compression is propagated as long as each processor along the path succeeds in updating its forwarding address.

Note that this code fragment is not made atomic. The atomicity of 'FAT$delete' and 'FAT$update' are all that are required to maintain the integrity of the local FAT. Lost messages and crashed processors can still cause a path compression as a whole to fail, but this is no problem because path compression is necessary only for efficiency, not for correctness. The correctness of the protocol depends only upon maintaining local consistency.

6.6 Remote Object Access.

As stated above, we are not specifying the semantics of object access as seen by a programmer using the system. At the level of our exposition an access is initiated by a processor sending an "Access Request" message that will eventually be delivered to a processor at which a handle for the object is located. We say "a processor" rather than "the processor" because it is possible that an object be in motion at the time that a processor initiates an attempt to access it.

6.6.1 Originating an Access.

Suppose that processor P wishes to originate an an "AccessRequest" message for an object named by CID 'x' and wishes to pass it 'argBlock', a data structure specifying the operation to perform on 'x' and an appropriate list of arguments for that operation. The APE at P should execute the following code fragment:

```
l: LMAPentry := LMAP$lookup(x)
if l ≠ NULL & l.reallyHere then
        {Access 'x' locally using 'l.handle'.}
        {Don't forget to check 'l.inMotion'.}
else ; Originate a remote access operation.
        f: FA := FAT$lookup(x)
        ar: Message := Message${dst: f.loc, src,origin: P,
```

```
                type: "AccessRequest",
                body: record${fa: f, args: argBlock}}
        Net$send(ar)
        end
```

Note that it may be that x is in the process of being moved to this processor. This is detected when there is an LMAP entry for x but 'reallyHere' is false. In this case the processor begins by treating the access as remote. If all of the relevant processors do not fail during the access then it will cause the move to either be commited or aborted. If it is commited then the "AccessRequest" will eventually be forwarded back to P. Otherwise, some other processor will handle it.

Because the network has not been assumed to be reliable it will be appropriate for the APE to put some or all of this code fragment inside a loop that implements a retry strategy.

6.6.2 Reacting to an "Access Request" Message.

A processor Q that receives an "accessRequest" message should perform the following action:

```
{ar := Net$receive() & ar.type="AccessRequest" } ⟶

    f: FA := FAT$lookup(ar.body.fa.name)
    if f = NULL then
        Net$send(Message${src,origin:Q, dst:ar.origin, type="AccessReply",
            body: record${status: "UnknownObject",
                currentFA: ar.body.fa}})
    elseif f.loc = Q then
            {Q has the object. Participate in object end of access.}
    elseif f.ts < ar.body.fa.ts then
            ; The object successfully moved to Q, but Q has not
            ; completed the transaction.
            registerFA(ar.body.fa, always)
            { Participate in object end of remote access.}
    else ; Forward the request.
        ar.src := Q ; Update 'ar' to reflect forwarding.
        ar.body.fa := f
        Net$send(ar) ; Pass it on.
    end
```

6.6.3 Completing an Access.

We are not making any specific assumptions about the implementation or the semantics of an access operation. The APE must implement mechanisms for matching replies with requests and for retrying requests that receive no response. It must specify the actual protocols used for transmitting information between the requesting and accessed objects and for synchronizing them.

We do require, however, that by the time that the operation is complete that the originating processor's OFP be informed of the location at which the object was found or the reason for the failure to find it. This information is conveyed in an "AccessReply" message. When the object is found this takes the following form.

```
ar: Message := Message${ src, origin: Q, dst: P, type: "AccessReply",
      body: record${status: "Success",
            currentFA: { an FA naming the current location },
            {information for the APE and application}}}
```

When P, the originator of an "AccessRequest", receives an "AccessReply" it performs the following action.

```
{m = Net$receive() & m.type="AccessReply"} ⟶

    obj: CID := m.body.currentFA.name
    g := FAT$lookup(obj)
    if m.body.status = "Success" then
        { Perform APE-defined actions for completing the access.}
        if g.loc ≠ m.body.currentFA.loc then
            if FAT$update(m.body.currentFA, true)
                ; This succeeds unless an update occured since the lookup.
                Net$send(Message${src,origin: P,
                        dst: g.loc, type: "PathUpdate",
                    body: m.body.currentFA})
                end
            end
        end
    elseif m.body.status = "UnknownObject" then
        ; 'obj' has been deleted.
```

```
    { Perform APE-defined actions for a deleted target object.}
    FAT$delete(obj)
    Net$send(Message${src, origin: P, dst: g.loc, type: "PathUpdate",
        body: FA${name: obj, loc: NULL, ts: NULL})
else ; Something exceptional has happened
   . { Handle the exception. }
end
```

This action has the effect of initiating a path compression if P discovers that the object is at a location other than where it was last seen or if the object has been deleted.

6.6.4 Other Remote Access Protocols.

The model implementation passes "AccessRequest" messages along a chain of forwarding addresses and later initiates a path compression by passing a "PathUpdate" message along that same path. This is but one of several alternative protocols to achieve the same end. We mention some of these here.

An alternative to forwarding an "Access Request" message when an object has moved is to use "WhereIs" and "Bulletin" messages. Suppose P wishes to access x and has a forwarding address for it that points to processor Q. P polls Q by sending it a "WhereIs" message. If the "Bulletin" message received from Q in response to this indicates that Q is still the location of x then the rest of the access is begun. Otherwise, P updates its forwarding address for x using the information in the "Bulletin" and repeats this process until x is found.

If the network is reliable and "AccessRequest" messages to not require acknowledgements at the level of the APE and OFP then this protocol doubles the number of messages used to find the object. In exchange, the originator knows which processor has the request. This has better reliability properties than the model implementation. If there is an appreciable probability that messages are lost then the originator knows exactly which processor to retransmit to. Furthermore if processors fail then the lack of response to a request can be used to help trigger the recovery of that node. Partial compression of forwarding address paths is possible even in the presence of failures. The

choice of which method to use depends upon the probability of failure. If the network and the processors are usually reliable then our model implementation can be used as the primary mechanism. It can be backed up with the other implementation implemented implemented using "WhereIs" and "Bulletin" messages.

One philosophical advantage to using "WhereIs" and "Bulletin" messages is that the act of successfully sending the "Bulletin" is that it guarantees that the sender will not attempt to use that particular forwarding address again. The sending of the "Bulletin" message thus splices the sender out of at least one path of forwarding addresses. Given that there is a "social contract" that obligates processors to participate in the object finding protocol, the sending of the "Bulletin" message is in a processor's "enlightened self-interest" because it decreases the amount of work that it is obligated to do in the future.

Yet another alternative form of path compression can be implemented by redefining the "AccessRequest" message so that it accumulates a list of the processors that have forwarded it. The processor currently holding the object is thus in a position to control the compression of the path. It can re-direct each processor along the path to point to it. This has good properties with respect to storage management and will be discussed in more detail in Chapter 8.

There are other techniques for shortening paths of pointers. Some of these are discussed in [TvL84]. Not all of them are suitable for use in managing forwarding addresses.

6.7 Summary

This model implementation is correct under our restricted model of failure. The protocol for moving an object guarantees that it is neither lost nor cloned as long as system failures are restricted to be either lost messages or processor crashes. Objects may be inaccessable because of processor crashes, either of the processor at which an object is located or of a processor on a forwarding address path. The postulation of the existence of stable storage together with the fact that forwarding addresses are kept there, however, guarantees that when the failed processors recover that the object will again be accessable. The set of

processors whose failure can make an object inaccessable is restricted to be a subset of those processors that it visits.

There are two major failings of the model implementation:

If an object x visits a processor then it may be necessary for it to keep a forwarding address for x for x's lifetime. Under our assumtions about reasonable system behavior this will not pose too much of a problem. We assume that long lived objects that visit a lot of processors will be very important and that the processors it visits will continue to want to access it as long as it exists. Nevertheless, in Chapter 7 we explore the modification of the implementation so that a processor that has held the representation can get rid of unwanted forwarding addresses to it.

The other problem is in the assumption of stable storage. This is a common but perhaps inadequate assumption. There will be some crashes from which processors will not be able to recover. In addition some applications will need uninterruptable access to distributed objects. In Chapter 8 that we will explore techniques for ensuring that distributed objects will continue to be accessible even in the presence of processor failures.

Chapter 7

Improved Storage Management and Administration.

In the model implementation of the previous chapter a processor at which an object has been located cannot delete its Forwarding Address Table (FAT) entry for it until it learns that the object no longer exists. This is because such a processor has no way of learning whether or not there are in fact any other processors depending upon the existence of that FAT entry.

In this chapter we discuss some extensions to the model implementation that allow a processor to discard those entries while still fulfilling its obligations to store other FAT entries and objects.

In our example of the Humongous Widget Corporation, we postulated an office automation system in which a HOPS (Humongous Object Programming System) object may represent a document. Early in its lifetime that object may make the rounds of several processors, but eventually the information in that document will be put in a database for long term storage. We postulated that the information would be transferred to a new object in the database and that the original would be destroyed. This allowed all of the processors that it had visited to delete their FAT entries for it. If instead of transferring the information to a copy the original is incorportated into the database then these processors would have no way of deleting those FAT entries.

In contrast with the problem of deleting FAT entries, the issue of deciding when

to delete objects is difficult. Problems arising from cycles of objects that mutually reference one another are compounded in systems such as ours in which it is possible that an object may be an active entity that may spontaneously awaken. We assume that the management of object storage is handled by the APE. It is outside the scope of the current work.

We define *passive storage management* to mean that for each data item at a processor there are certain locally defined conditions that inform that processor whether it is allowed to recover the storage of that data item. The processor must, however, wait passively for for those conditions to be satisfied. It does not have the means to force them to become true. Another way of expressing this is that while the processor has certain obligations with respect to the use of its storage, it cannot actively attempt to discover them, nor can it renegotiate them. It must passively wait for other processors to inform it as to the status of those obligations.

In comparison we use *active storage management* to mean that a processor has the ability to perform an action which, if successful, guarantees that it can recover a particular piece of storage. In other words, it has the ability to actively renegotiate its obligation to store a particular data item. In our context this means that a processor P mav have the ability to force an object in which it has no further interest to move to some other processor. Furthermore, it means that P can force all other processors to update their FAT entries for an object not at P so that P can get rid of its own FAT entry for that object.

If the system provides a mechanism that allows the active management of FAT entries and objects then processors can actively administer the allocation of their own resources. Dannenberg's work on the Spice Butler [Dan82, DH85] is an exposition of one approach to these issues. With these mechanisms processors can negotiate among themselves as to where resources should be. Furthermore, processors can negotiate their way out of specific applications. This is useful not only for controlling the allocation of resources, but it means that the system can be reconfigured in an orderly way under administrative control. Because of our assumptions about the degree of administrative autonomy of

parts of very large decentralized systems we believe that the ability to actively manage a processor's resources is very important.

A surprising result of our work is the conclusion that for forwarding addresses a mechanism supporting active storage management can be more efficient and easier to implement while providing the same level of reliability than a mechanism for passive management.

7.1 Passive Storage Management.

The classical methods of storage management are reference counting schemes and garbage collection. The arguments in favor of garbage collection over reference counting usually take two forms:

- If the objects of interest are small then reference counting can be comparatively expensive. If the objects are very small then the addition of a reference count to its representation is a considerable burden.
- Reference counting schemes are not able to recover cycles of passive items that reference each other but which are not accessible from outside the cycle.

In the context of managing the storage of FAT entries the first objection is not important since a FAT entry will be many times larger than the size of an integer needed to count references to it.

The second objection is similarly not a problem for forwarding addresses. In the model implementation for atomic objects each forwarding address contains a pointer to a single other processor. As long as the object referenced is not in motion then the graph of forwarding addresses will be acyclic except for the self-loop at the processor at which the object is located. With this exception the graph of forwarding addresses is a tree.

In the generalized case in which an object is distributed each forwarding address will point to a set of processors at which the object was simultaneously located. (This will be discussed in detail in Chapter 8.) The processors at which the object is currently located will all point to each other. The monotonicity of the timestamps along a path

of forwarding addresses once again guarantees that the graph will be acyclic with the exception of the strongly connected component of the object's current location.

Since the forwarding addresses at the current location of an object are guaranteed to be useful, we do not have to worry about not being able to recover the storage of a useless forwarding address because it is in a cycle.

In contrast to the potential problems with reference counting, the big disadvantage of using garbage collection as a storage management mechanism is that it is potentially global in scope. The cost of determining whether there is a processor that possesses a name for an object or a forwarding address that points to a particular processor increases with the number of processors that might. Furthermore, as the number of processors rises the probability that at least one of these processors will be unavailable at any time also rises. For these reasons we do not believe that a general garbage collection scheme will be viable on a very large distributed system. It may, however, still be possible and desirable within a single local area network or other well defined sub-systems.

Based solely on these arguments the use of reference counts for managing forwarding addresses appears to be potentially worthwhile. Once we begin to look at possible implementations and to compare them with the model implementation of Chapter 6 we begin to see problems. The primary among these is the increased complexity of the protocols for modifying the FAT.

The registration of a newly arrived forwarding address can entail a pair of two-processor transactions. Because reference counts must be guaranteed consistent one transaction is necessary to stably store the newly arrived forwarding address and to simultaneously and reliably increment the reference count at the processor to which it points. A second transaction is necessary to simultaneously and reliably delete the old forwarding address from stable storage and decrement the corresponding reference count. Two separate transactions seem preferable to a single three-processor one because the two functions being performed are logically separate. The second transaction can be performed as part of the path compression protocol and can be run at a lower priority than the first one.

This necessity of using multiple processor transactions to correctly maintain reference counts multiplies the cost of making a modification to the forwarding address graph by a factor of as much as eight. Changing a FAT entry in the model implementation of Chapter 6 involved changing the FAT only at the processor holding the forwarding address. This involved writing to stable storage once. If we use reference counts then the change entails modifying the reference counts for the FAT entries at both the processor pointed at by the old forwarding address as well as the processor pointed at by the new one. Since each of these involves a two–processor transaction and since each processor in a transaction writes to stable storage at least once in each of two phases, a straightforward implementation of reference counts would cost about eight times as much as our model implementation of the simple protocol. This factor can be reduced by replacing the two–processor transactions with larger ones. For instance, the pair of transactions implementing a single stage of path compression can by replaced by a single three–processor transaction. If each processor in the three processor transaction no more than once during each phase of the protocol, then the total number of writes is reduced from eight to six. By turning several stages of path compression into a single transaction the factor could be reduced further. For a very long path that is compressed in a single transaction the factor asymptotically approaches two. This negates, however, the considerable advantages of being able to incrementally and concurrently modify the forwarding address graph.

Because the passing and registering of a forwarding address entails the added expense of the transaction to increment a reference count it makes little sense to register any forwarding addresses other than one pointing to an object's current location. This adds an additional traversal of a forwarding address path for each time a processor receives a new forwarding address.

All–in–all, the use for reference counts for the passive management of the storage of forwarding addresses is comparatively expensive and difficult. The disadvantages of using reference counts will become more apparent in the discussion of active storage management.

7.1.1 Counted Forwarding Addresses as Objects.

If, despite our negative comments about the use of reference counted forwarding addresses, one still feels compelled to use them we offer the following suggested implementation. This scheme is an adaptation of a technique proposed by Gifford [Gif82].

The reference count for an entity is the cardinality of a set of objects of a certain kind, the set of existing references to it. This count must be guaranteed to be correct. This can be guaranteed if both the count and the objects are kept reliably in stable storage.

We start by assuming the existence of the model implementation of Chapter 6. We augment it with two other notions of forwarding address. *Ordinary* forwarding addresses are those of the model implementation. These are also called *uncounted* forwarding addresses because they convey no guarantees as to their usefulness. To form a *counted* forwarding address we encapsulate an uncounted one within an object created specifically for that purpose. The reference count kept by processor P for each object is the number of existing counted forwarding addresses for that object that point to P.

To ensure that the reference count is accurate, an object encapsulating a counted forwarding address can be created and destroyed *only* at the processor to which it points. Thus the creation of a counted forwarding address and the incrementing of the corresponding reference count can be done simultanesously within a local atomic action. Similarly, another local atomic action can both destroy the object and decrementing the count.

Uncounted FA's can be created by copying the contents of counted ones. They can be used to find and access the objects they name, but they convey no guarantee as to their usefulness. A processor can obtain that guarantee only by obtaining a counted FA. This can be done either by moving one from another processor or by requesting that the processor pointed to create a new counted FA and send it back. It relinquishes the guarantee by sending the counted FA to some other processor. That processor is then free to destroy it and decrement the corresponding reference count.

The attraction using this approach to implementing reference counts is that it takes

advantage of the existence of the simple protocol for reliably moving an object, thus obviating the need to create a special form of distributed transaction for incrementing and decrementing reference counts.

Since the counted FA's are themselves objects that are findable using ordinary FA's, the processor they point to could retain a list of their CID's and could track them down. For example, it could use this ability to request that a particular counted FA object be returned so that it can be destroyed. A processor can thus obtain a degree of active control over its obligations to other processors.

Because this implementation is based on the idea of reference counts it is still overly cumbersome. It entails the use of object moving transactions to manage the counted FA's and the use of ordinary FA's to keep track of them. We will see in the next section that this is overly complicated and unnecessary.

7.2 Active Storage Management: Control of Obligations.

When the 'seenHere' flag in a FATEntry is set to true it means that the processor holding that entry is obligated to keep it because there may be some other processor that depends upon it. A reference count means that there are exactly that number of outstanding obligations, but the identities of the processors (or objects) to which they are owed are unknown.

Both of these represent nebulous commitments of a kind not generally acceptable outside of computer systems. For example, suppose that you run a warehouse. Furthermore, suppose someone stores a physical object such as a trunk in your warehouse. What sorts of obligations will appear in the contract to store the object? First, the customer agrees to pay some amount to store the object in return for which you agree to keep and protect it. The contract will be for some specified time. Within that time period the warehouse agrees to deliver the object on demand to the individual that stored it or to an individual bearing the original contract. If, while the object is being stored, the original owner wishes to transfer its ownership, all of the obligations of the warehouse do not automatically get transferred. The seller has a contractual obligation to

deliver the object to the buyer that is distinct from the warehouse's obligation to the seller. To establish a direct obligation of the warehouse to the buyer the warehousing contract must be renegotiated in a separate but related transaction. The warehouse's obligations cannot be arbitrarily reassigned to other individuals by a third party without its knowledge.

If during the active period the buyer of the object shows up with the warehousing contract then he may retrieve it or renegotiate a new contract for its storage. This is evidence of a transfer of ownership. Consider, on the other hand, what happens if the object has not been reclaimed at the end of the contract period? Even though the contract has expired, you as the warehouse owner will probably not have the right to immediately and arbitrarily dispose of the object. You may be under an obligation to expend a reasonable effort for a reasonable length of time to find the other party to the lapsed contract. Because the contract explicitly names its parties this is easy to do. If the original owner of the object has sold the rights to it and then disappeared, however, there is no way of knowing that he has sold it, or to whom.

In the real world we generally do not have obligations to specific, but unknown individuals. We know what obligations we owe and to whom they are owed. Obligations cannot be arbitrarily transferred and multiplied. In light of this observation, we consider a form of active storage management in which the obligations of a processor are represented explicitly.

7.2.1 Reference Sets.

The 'seenHere' field of a FATEntry in the model implementation can be interpreted as meaning that there is a possibly non-empty set of processors that depend upon this FATEntry and for that reason the processor is obligated to keep it. Similarly, a reference count can be interpreted as the cardinality of that set. The next step in this progression is to actually store at each processor the identifiers of the set of processors that could depend upon the existence of the FATEntry. This set of processor identifiers represent back pointers in the forwarding address graph. We call it the *reference set* at that

processor for the object referenced. Halstead [Hal79] used such back pointers in his reference trees.

The use of reference counts requires that the processors at each end of an edge in the forwarding address graph communicate with each other in a multi–processor transaction each time an edge is created or destroyed. Because that communication and updating of stable storage is already necessary to maintain reference counts there would be little added communication and computation cost to keep reference sets. The main penalty would be to approximately double the total amount of local storage devoted to finding objects in the worst case.

While we believe that this is a relatively small price to pay to obtain active administrative control of the system, it may still present difficulties. Consider an object that is widely known to be at a particular processor. The reference set for that object may be huge, occupying much more storage at that processor than the representation of the object itself. It may be economical to handle such instances as a special case.

7.2.2 Inexact Reference Sets.

The reason that multi–processor transactions are necessary for the maintenance of reference counts is that they must be kept exactly. The processor can delete the corresponding entity only when the count goes to zero. If it is accidentally incremented when it shouldn't be then the count will never be decremented to zero and the entity can never be destroyed. If it is accidentally decremented when it shouldn't be then the count may go to zero prematurely and the entity will be prematurely destroyed. The reference count must be kept accurately and reliably because the processor holding the entity has no direct knowledge of to whom it is obligated. In has no method of actively determining the extent of its obligations. Instead it can only wait passively, incrementing and decrementing the count until it goes to zero.

If reference sets are used a processor can take active steps to reduce its obligations. Because storage management can be done actively there is little problem if the reference set for x at P contains the identifier of a processor Q to which P does not have an

obligation. P can poll Q to find out whether or not the obligation in fact exists and if so to renegotiate it. P can still manage its storage as long as the reference set that P maintains is a *superset* of the set of identifiers of processors to which it does have obligations. Since the reference set and the outstanding obligations no longer have to be in exact agreement protocols that use them can use weaker but more efficient methods than multi–processor transactions to ensure correctness. As in Chapter 6 we are assuming that each processor has local stable storage that will survive any processor crash.

Active storage management can have additional advantages. When a processor is given the ability to actively manage its storage it gains the power to reconfigure itself. This power to reconfigure can be used for administrative reasons as well as in response to failure.

In the remainder of this section we explore the use of inexact reference sets by extending the model implementation of Chapter 6. This is the extended definition of an entry in the forwarding address table.

```
FATEntry = record[
            fa: FA,
            refSet: Set[ProcID],
            guarantors: Set[ProcID],
            gLock: int].
```

We assume that the abstract data type 'Set' implements mathematical sets in such a way that all the usual operations on sets can be denoted using conventional mathematical notation. In extending the basic model implementation we replaced the Boolean flag 'seenHere' with two sets of processor identifiers ('ProcID's). The set 'refSet' names a *superset* of the set of processors which depend upon the existence of 'FATEntry' for the object at this processor and to which this processor is thus under obligation. The cardinality of 'refSet' is thus at least as large as a reference count would be. The set 'guarantors' names a *subset* of the processors that believe themselves to be obligated to this processor to keep a FATEntry for the object.

Our definition of "obligation" is that Processor P is obligated to processor Q to keep a 'FATEntry' for object x if and only if the forwarding address in Q's 'FATEntry' for x points to P.

To ensure that 'refSet' is indeed a superset of the processors to which P is obligated a 'ProcId' can be added to the 'refSet' at any time, but in particular it must be added no later than the establishment of an obligation to that processor. To ensure correctness, the attempt to write an entry into Q's FAT pointing to P is begun only after it is known that the local atomic action at P that adds Q to P's 'refSet' for Q has completed successfully. Because exact agreement is not required there is no need for a multi–processor transaction to perform this pair of actions simultaneously and reliably.

A processor Q can be removed from a 'refSet' no earlier than the release from the obligation at to Q. A multi–processor transaction is not required to do this, but there is a need for synchronization to prevent an attempt to release P from an obligation to Q at the same time that it is possible that there is simultaneously an undelivered message in the network which, when delivered, will re–establish the same obligation. We use the assumption that the messages from any processor to any other are received in order (if at all) to establish that synchronization. The field 'gLock' (an abbreviation of "guarantee lock") is a counter used to distinguish different requests from P to confirm the release of an obligation. While 'gLock' has a positive value P may not extend any new guarantees about that FATEntry. Furthermore, it will also react only to "ReleaseObligation" messages for x that mention that specific value. While 'gLock' is not positive P can extend all the new obligations it wants to but it cannot act on any "ReleaseObligation" messages at all. The absolute value of 'gLock' is a count of the number of times that P has begun a phase of trying to get rid of its obligations towards this 'FATEntry'.

We anticipate that in a real system that the need for such synchronization will be rare. As we have written the pseudo–code, a processor never extends a new guarantee for an object unless it is in fact the current location. A reasonable processor will not often be simultaneously attempting to distribute forwarding addresses that point to itself at the same time that it has decided that the object is not interesting and is attempting to

reclaim the storage occupied by its 'FATEntry'. It would instead wait until the object had moved elsewhere and its "interest" in it had waned before attempting to be released from these obligations. Nevertheless, these synchronization problems can arise and must be dealt with. Note that a determined attempt to reclaim a 'FATEntry' for an object not currently at P will usually succeed so the number of bits necessary to represent 'gLock' will be small.

To make sure that 'guarantors' names no processor that is not obligated to Q the identifier for P cannot be added until the obligation is in fact established by storing a forwarding address pointing to P. P may be removed from 'guarantors' at any time, but it must be removed before P learns that it has been released from the corresponding obligation represented. 'Guarantors' is included here stricly for the purposes of efficiency. The correctness argument for the protocol is consistent with 'guarantors' always being empty. It could therefore be removed from the definition.

The Forwarding Address Table itself is redefined as:

```
FAT = stable cluster is create,entRead,entWrite,delete,lookup,elements

    rep = { an appropriate concrete representation }

    own fat: rep

    create = proc() returns()
            {Create and initialize the table 'fat'.}
            end create

    entRead = proc (x: CID) returns(FATEntry)
            { Read the entry for 'x' and if it exists return a copy.
            Otherwise return NULL. }
            end entRead

    entWrite = atomic proc(fe: FATEntry)
            { Write 'fe' to the FAT, replacing any old entry
            for 'fe.fa.name'.}
            end entWrite

    delete = atomic proc(x: CID)
            {Delete the entry for 'x' from the FAT.}
```

```
        end delete

    lookup = proc(id: CID) returns(FA)
        {Return the forwarding address for 'id' if it is in the FAT.
        Otherwise returns the NULL forwarding address.}
        end lookup

    elements = iter () yields(FATEntry)
        ; Returns each of the elements in the FAT.
        j: FATEntry;
        for j ∈ fat do
            yield(j);
        end
      end elements

    end FAT
```

Because of the added complexity of using reference sets the operations 'add' and 'update' have been replaced with 'entRead' and 'entWrite'. The functionality of 'update' has been moved to a higher level.

Local manipulations of objects are mostly unaffected by the addition of reference sets. The 'baptise' procedure is modified to initialize 'refSet', 'guarantors', and 'gLock'.

```
baptise = atomic proc(obj: OHandle) returns(CID)
    c: CID := CID$create() ; Get a new CID for the object.
    t: TimeStamp := 0 ; Get a zero TimeStamp.
    if LMAP$insert(LMAPentry${name: c, ts:t, handle: obj,
                    reallyHere: true, inMotion: false})
    then
        FAT$entWrite(FATEntry${fa: FA${name:c, loc: P, ts: t},
                    refSet, guarantors: φ,gLock: 0});
        return(c)
    else fail ; Couldn't baptise, handle as exception.
    end
end baptise
```

As before, we make no assumptions as to when an object can be destroyed. The existence of reference sets may be useful to the Application Programming System (APE)

for managing object storage. If an object x is located at processor P then the 'refSet' for x at P can be used by P to decide whether destroying x could leave any dangling references for it. If the 'refSet' at P is empty or can be made empty by getting the processors in it to release P from its obligations then the APE at P is free to deal with x as it pleases. Even if this condition is not obtainable the APE may be able to run a garbage collection protocol for cyclic structures localized to the processors in the transitive closure of the 'refSet' for x at P obtained by including its elements together with elements of the 'refSet's for x at all processors named in that closure. Thus, the garbage collection of an object in a cycle can be localized to the set of processors that actually have references for it.

The protocol for moving an object is unchanged except for ensuring that the new and old locations are included in each other's 'refSet' no later than the end of the the first phase of the two–phase commit protocol. Both processors can also add the other to their 'guarantors' set at the appropriate times.

The sending and receiving of forwarding addresses is modified to reflect the fact that when initially received a forwarding address may not be guaranteed to be useful for finding an object.

When a CID (or forwarding address) is sent in an application message to another processor the sending processor P should guarantee the forwarding address if it is the current location of the object. It must also check to see if it is attempting to reduce its obligations.

{CID 'x' appears in an outgoing application message 'm' to Q.} $\longrightarrow$

```
atomic begin
    fe : FATEntry := FAT$entRead(x);
    if fe.fa.loc = P then
        if fe.fa.gLock > 0 then
            fe.fa.gLock := −fe.fa.glock ; Break the lock
            end
        fe.refSet := fe.refSet ∪ {Q}
        FAT$entWrite(fe)
        end
```

```
    end ; of the atomic action
{Append 'f' to the end of 'm'.}
```

In this implementation we have assumed that sending the application message takes absolute priority over the ongoing effort to reduce the processor's obligations. It therefore negates 'gLock', breaking the lock against extending new guarantees and allows the message to be sent. Other strategies such as one that waits a reasonable time for the obligation reduction phase to complete could be used instead.

When a forwarding address is received in a message it may be necessary to get a guarantee as to the usefulness of a forwarding address before it is stored in the FAT. The procedure 'registerFA' is modified as follows.

```
registerFA = atomic proc(f: FA, always: Boolean) returns(Boolean)
    ; Return true iff the FAT is modified.
    ; Return false iff the FAT is not modified.
    ; Fail if no FAT entry can be created.
    lme: LMAPentry := LMAP$lookup(f.name)
    fe: FATEntry := FAT$entRead(f.name)
    if fe = NULL then ; Build a real one
        fe:= FATEntry${fa: NULL, refSet, guarantors: φ, gLock: 0}
        end
    if (lme≠NULL & f.ts≥lme.ts& ¬lme.reallyHere) then
        ; A move to Q is in progress and 'f' can complete it.
        if f.loc = Q then ; Q is the ID of this processor.
            ; Commit the move.
            lme.reallyHere := true
            lme.ts := f.ts
            LMAP$update(lme)
            fe.fa := f
            { Optionally remove some elements from fe.guarantors. }
            FAT$entWrite(fe)
            return (true)
        else ; Abort the move
            LMAP$delete(f.name)
            fe.fa := f
            FAT$entWrite(fe)
            return (true)
            end
```

```
elseif lme = NULL then
    ; an ordinary update of the FAT.
    if fe.fa = NULL & ¬ always ∨
            fe.fa ≠ NULL & f.ts < fe.fa.ts then
        ; There was no old entry and no imperitive to create one, or
        ; No update because f is older than fe.fa is old
         return(false) ; No entry in FAT.
        end
    if f.loc = m.src ∨ f.loc = fe.fa.loc ∨
            f.loc ∈ fe.guarantors then
        ; If m was received from either f.loc or fe.fa.loc, or
        ; f.loc is in guarantors,
        ; then the guarantee is already established.
        fe.fa := f
        FAT$entWrite(fe)
        return (true)
    else ; It is necessary to get a guarantee for f.
        f2: FA := getGuaranteedFA(f)
        if f2 ≠ NULL then
            fe.fa := f2
            { Optionally add f2.loc to fe.guarantors. }
            FAT$entWrite(fe)
            return (true)
        else ; No guarantee was obtained.
            if fe.fa = NULL then fail
            else
                return(false)
                ; Even though no new guaranteed FA could be gotten
                ; the old one is still usable.
                end
            end
        end
    else
        { Q has an LMAP entry for 'f.name'.
        Treat it as an exception.}
        end
    end registerFA
```

This is a long but straightforward extension of the 'registerFA' of Chapter 6. Note that in this version 'registerFA' can fail because it is unable to get a guarantee for a newly arrived FA. This occurs when Q did not have an existing 'FATEntry' for the object and

it was unable to get a guarantee on the new one. At this point Q should attempt to access the object using the unguaranteed forwarding address. Reasons for a failure to guarantee 'f' include that the object itself has been destroyed and the possibility that the processor that originally had 'f' did not have it guaranteed or allowed the guarantee to be released. Whatever the reason for the failure to guarantee 'f'. The problem must be resolved at a level higher than the Object Finding Package. The APE at Q should have a further dialogue with the processor that gave it 'f' to resolve this problem.

The procedure 'getGuaranteedFA' as executed at Q is defined as follows.

```
getGuaranteedFA = proc(f: FA) returns(FA)
    while true do
        Net$send(Message${src,origin: Q, dst: f.loc,
                type: "GuaranteeRequest", body: f})
        select ; Wait for one of the guard conditions to become true.

            {m:=Net$receive() & m.type="GuaranteeReply"
                & m.body.fOld=f} ⟶
            begin ; This is a reply to our message.
                if m.body.fNew.loc = m.src then ; Success
                    ; m.body.fNew.loc is a guaranteed FA.
                    ; it must be more recent than f.
                    return(m.body.fNew.loc)
                else ; Failure
                    return(NULL)
                end ; of this action

            { The message times out. Try again. } ⟶ continue

            { Give up. } ⟶ return (NULL)
        end ; the select statement
    end ; the while loop
end
```

A processor P responds to a "GuaranteeRequest" using the following protocol.

```
{m = Net$receive() & m.type = "GuaranteeRequest"}⟶

    atomic begin
        fe: FATEntry := FAT$entRead(m.body.name)
```

```
if fe = NULL then
    ; Never heard of it. Can't help guarantee it.
    Net$send(Message${src,origin: P, dst: m.origin,
        type: "GuaranteeReply",
        body: record${fOld: m.body, fNew: NULL}})
bf elseif fe.fa.loc ≠ P then
    ; The object is not here.
    ; Forward the request.
    m.src := P
    m.dst := fe.fa.loc
    m.body := fe.fa
    Net$send(m)
else ; Guarantee the request.
    if fe.gLock > 0 then
        fe.gLock := -fe.gLock ; Guarantee has priority, break lock.
        end
    fe.refSet := fe.refSet ∪ m.origin
    FAT$entWrite(fe)
    Reply with the latest FA.
    Net$send(Message${src,origin: P, dst: m.origin,
        type: "GuaranteeReply",
        body: record${fOld: m.body, fNew: fe.fa}})
    end
end ; of the atomic action
```

While it would be correct for P to reply positively to any valid request to guarantee a forwarding address, in this implementation it does so only if the object it names is located at P. Otherwise it forwards the request using the current entry in its FAT. The procedure 'getGuaranteedFA' looks at the reply to its request and if the forwarding address it receives points to the same processor that sent it then it knows that the returned forwarding address has been guaranteed. It passes it back to 'registerFA' which installs it in the local FAT. Failure can be the result either of receiving a NULL forwarding address meaning that no one will guarantee it or as a result of waiting for too long and giving up on the attempt. As mentioned above the refusal to get a processor to guarantee a forwarding address means either that the object has been destroyed or that the forwarding address mechanisn has been misused. In either case the resolution of the

problem must be done at a higher level in the system.

One way of looking at a successful attempt to guarantee a forwarding address is that it is a kind of access operation on the object named. The protocol for passing the "GuaranteeRequest" message is exactly the same as that for passing an "AccessRequest". As we have written the example a "GuaranteeRequest" does not trigger a path compression, but it could.

While the remote access and path compressing primitives of Chapter 6 can be used in unmodified form we note that since the processors along the path will eventually all point to the current location of the sought object that they will all eventually have to communicate with that processor in order to guarantee their newly acquired forwarding addresses. If the length of the path is k then if the current location of the object distributes guaranteed copies of the most up to date forwarding address for it then the number of messages needed to compress the path is $k-1$. On the other hand, if we use the incremental path compression strategy combined with the "GuaranteeRequest" protocol sketched above, the number of messages used is $(k-1)(k-2)/2$. For this reason the variation of path compression in which the access request message accumulates the identifiers of the processors it encounters on its travels should probably be the one used in conjunction with reference sets.

One thing that the implementation of Chapter 6 did not have is the notion of releasing of a processor from an obligation. If processor P wishes to reduce its obligations with respect to a particular object it can execute the procedure 'beginReleasePhase' to begin a phase of attempts to reduce them.

```
beginReleasePhase = proc(x: CID)
    fe: FATEntry
    lock: integer
    atomic begin
        fe : FATEntry := FAT$entRead(x);
        if fe = NULL ∨ fe.gLock > 0 then
            ; Either no obligations or a phase is already in progress.
            return
        lock, fe.gLock := −fe.gLock + 1 ; Set lock for a new phase.
        FAT$entWrite(fe)
```

```
    end; of the atomic action
for q: ProcID ∈ fe.refSet do
    Net$send(Message${src,origin: P, dst: Q, type="ReleasePhase",
            body: record${ currrentFA : fe.fa, phase: lock })
    end
end
```

Note that the "ReleasePhase" message P sends contains its forwarding address for x rather than just x itself. This is to promote the active reduction of P's obligations. If x is not at P then the forwarding address contains the information the destination of the message needs to release P from its obligation.

When processor Q receives a "ReleasePhase" message it can try to reply constructively as follows.

```
{m := Net$receive() & m.type = "ReleasePhase"}⟶

    atomic begin
        x: CIR := m.body.name
        fe: FATEntry := FAT$entRead(x)
        if myFA = NULL ∨ myFA.loc≠m.origin then
            ; There was no genuine obligation.
            if fe ≠ NULL then
                ; Remove m.origin from fe.guarantors.
                ; '\' denotes set difference.
                fe.guarantors := fe.guarantors \ {m.origin}
                end
            ; The release will be successful
        elseif myFA.loc=m.origin & m.body.f.loc ≠ m.origin then
            ; There's a current obligation that might be removed.
            if registerFA(m.body, false) then
                fe.guarantors := fe.guarantors - {m.origin}
                ; The release will be unsuccessful
                end
            else ; Unable to remove the obligation.
                ; The release will fail.
                ; It may be useful to do something here.
                end
            end
        ;Send a release message containing Q's current FA in any case.
        Net$send(Message${src,origin: Q, dst: m.origin, type="ReleaseFA",
```

```
                body: record${f: fe.fa, phase: m.body.phase}
end ; of the atomic action.
```

In this action Q first determines whether there is in fact an obligation to the sender of the "ReleasePhase" message. If so then it tries to remove that obligation by registering the forwarding address sent in the message. The reply to a "ReleasePhase" message is a "ReleaseFA" message whose body contains Q's final forwarding address for x together with the phase number from the "ReleasePhase" message it is responding to. If the forwarding address points to 'm.origin' then the obligation has not been released. Otherwise it has been.

Given the ability to actively reduce its obligations processor P can eventually get rid of the FAT entry. It can check to see if it can do this each time it receives a "releaseFA" message.

```
{ m := Net$receive() & m.type = "releaseFA"}⟶

    atomic begin
        if m.body.f.loc = P then
            ; The obligation was definitely not released.
             Make note of this and perhaps begin a negotiation to move
        else
            fe: FATEntry := FAT$entRead(m.body.f.name)
            if fe = NULL then
                ; If we're here then we've already thrown away the FATEntry.
                {Handle this exceptional condition.}
            elseif m.body.phase = fe.gLock then
                ; Remove the comitment from the refSet.
                fe.refSet := fe.refSet \ {m.src}
                if fe.refSet = φ & ¬possiblyKnownLocally() then
                    ; The entry is deletable
                    FAT$delete(m.body.f.name)
                else
                    ; Save the modified entry.
                    FAT$entWrite(fe)
                end
            end
        end
```

7.2.3 An Informal Correctness Argument.

The key to the correctness of the use of inexact reference sets to control the obligations of each processor with respect to each object for which it has a forwarding address lies in establishing the invariant that the reference set is at all times a superset of the set of processors that depend upon the existence of that forwarding address. If the guarantors set is ever not empty then it is also necessary to establish the invariant that if a processor is named in a 'guarantors' set for an object then it is obligated to keep its FAT entry for x by virtue of the fact that P is named in its 'refSet'.

In our partial pseudo–code implementation no forwarding address that points to P can be stored in processor Q's FAT until P has first added Q to P's 'refSet'. This can be verified by inspecting the 'registerFA' and 'getGuaranteedFA' procedures and the code fragment that reacts to the receipt of a "GuaranteeRequest" message. The protocol for moving an object also establishes this condition. All forwarding addresses appearing in FAT's are therefore guaranteed.

Once appearing in the 'refSet' the only way that Q can be removed is in the action responding to the receipt of "releaseFA" message to P from Q. The precondition for the sending of this message is that the obligation has in fact been released.

When P receives the "releaseFA" message it knows that the obligation had been released, but it does not know whether or not there might be undelivered messages in the system that could restore the obligation without P's knowledge.

To prove that there cannot be we first note that the obligation cannot be re–established without a message sent from P arriving at Q. This may be verified by inspecting the pseudo–code implementation. Furthermore, recall that we have assumed that the network will deliver the messages from P to Q in order although it may lose some. If the "ReleaseFA" message was in response to a "BeginPhase" message whose guarantee lock ('gLock') has not been broken then the following conditions must be true:

- Any messages carrying guarantees that were sent from P before the sending of the "ReleasePhase" message must arrive at Q before it if at all. The atomic action at Q that actually ensures that the obligation is gone begins after the receipt of this message. These earlier messages cannot restore the obligation.

- If the guarantee lock is still in place inside the atomic action executed in response to receiving a "releaseFA" then we know that no new guarantees could have been extended in the time interval between the setting of the lock and when we exit from the atomic action.

Together, these two conditions imply that if the guarantee lock is still set then P's obligation to Q can not be restored by a message from P to Q that was sent prior to the enclosing atomic action. This is illustrated in Figure 7.1. It is therefore safe to remove Q from the 'refSet'.

7.3 Discussion.

The jump from the simple model implementation of Chapter 6 to doing passive storage management using reference counts is fairly large. It entails the use of a multi–processor transaction every time an edge in the forwarding address graph is modified. In addition to the increased computational burden it greatly increases the number of times stable storage must be written. From there going to active storage management using exact reference sets requires only the use of some additional memory to represent the back pointers in the graph. There is no additional communication load. The administrative benefits of having active storage management make the use of reference counts to keep track of forwarding addresses seem to expensive in relation to what you get.

The use of inexact reference sets further increases the amount of local storage needed since each processor stores not just backpointers, but potential backpointers. The fact that the reference set is kept inexactly means that we can eliminate the need to use multi–processor transactions, relying instead on a more primitive and efficient protocol that relies upon the ordering of local atomic actions and of message delivery. Because the inexact reference set allows a processor to take an active part in the administration of its storage, the inconsistencies between the inexact reference set and the obligations it represents can be efficiently discovered and eliminated. Even so the protocol is more expensive than the simpler implementation.

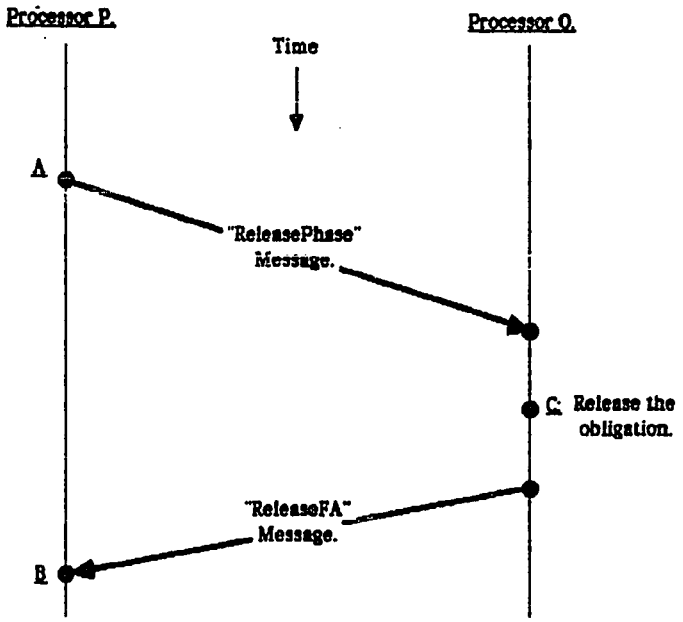

Figure 7.1: A phase of attempting to release obligations. Guarantees made by P before the beginning of phase j must arrive before the "ReleasePhase(j)" message. No new guarantees are extended during the phase. Therefore the obligation has really been released and no undelivered message can re–establish it.

The enhancements to the basic forwarding address mechanism discussed in this chapter are of interest only for a specific, perhaps small, subset of the objects and processors in the system. These are the very long lived objects that move among processors that do not keep a long term interest in them. The enhancements will not help other objects or processors.

To eliminate the cost of having to establish guarantees for short–lived objects we could resurrect the 'seenHere' flag in the FATEntry together with a 'alwaysGuaranteed' flag in a forwarding address. The meaning of the latter flag is that if it is true then the processor pointed to guarantees to keep its FATEntry until the object is deleted. The 'seenHere' flag represents that guarantee. Thus, within the inexact reference set protocol 'seenHere' can be interpreted as representing the universal set of processors. Not only is this more efficient for short lived objects, but in the case in which the object is extensively known it relieves the processor hosting the object from the burden of keeping a very large reference set for it.

Chapter 8

Enhanced Reliability and Distributed Entities.

The model implementations of forwarding address mechanisms that we have discussed thus far have had the following reliability properties.

- The processor model upon which they are based assumes that the only processor faults are *crashes* and that there is a class of storage that is *stable* in the sense that it is neither corrupted nor destroyed by a crash. While a crash may make entities stored in stable storage temporarily unavailable, we assume that processors eventually recover from a crash with its stable storage intact. This reliability model allowed us to assume that processors can execute local atomic actions.

- Our model implementation of the basic forwarding address mechanism relies upon the existence of local atomic actions for its correctness. Processors may crash making certain objects inaccessible, but after they recover the graph of forwarding addresses for each object will be in a consistent state. The storage management enhancements of adding reference counts or exact reference sets depend upon using simple two-processor atomic transactions. These can be implemented using the *move* protocol of the basic mechanism.

- These protocols do not depend upon reliable message delivery for their correctness. They do require, however, that although messages may be lost that the messages from any processor to any other that are delivered are received in order. We have assumed that the underlying system implements a logical network that appears to be completely connected. This logical network provides adaptive point-to-point message routing facilities. Because our protocols provide their own "end-to-end" [SRC84] consistency the underlying system need not be completely reliable. The network can therefore choose to provide only that level of reliability that it can implement efficiently.

- The accessibility of object x from processor P can be affected only by failures of the network to deliver messages or by the failure of a processor at which x has been located since P last accessed it. The more often a processor accesses an object the less probable it is that an attempted access could be affected by the failure of a processor on a forwarding address path. Each application is sensitive only to failures of the processors that participate in it.

For many applications these properties will be sufficient. Others may require enhanced reliability and accessibility properties. In this chapter we extend the basic forwarding address mechanism for atomic objects so as to provide enhanced accessibility for distributed objects. If an object itself has enhanced reliability then we will extend the forwarding address mechanism to match.

One form of enhanced reliability requirement is that a distributed object be accessible from any functioning processor even though an arbitrary set of up to k processors may have crashed and not yet recovered. Another enhancement is to strengthen the fault model by adding "fail–stop" failures. This model allows a processor to have a crash from which it never recovers. We will call this a *disaster*. A disaster is considered to start at some particular event and to last for ever without recovery. The application may require that a functioning processor still be able to access an object even if k disasters have occured.

To capture some appropriate notions of improved fault tolerance we will say that an entity is k–fault accessible if it can continue to be accessible from any non–failed processor in the presence of any combination of no more than k simultaneous crashes and disasters.

8.1 Reliability at Other Levels of the System.

It is our goal to illustrate forwarding address based object finding mechanisms that provide k–fault accessibility For this to be meaningful we require that other levels of the system be able to tolerate k–faults.

8.1.1 Reliability and the Logical Network.

The logical network implemented by the underlying system needs to be implemented so that even in the presence of k failures it can still transmit messages between any pair of functioning processors that appear at the ends of an edge in a k–fault tolerant forwarding address graph.

We assume that the logical network uses a routing algorithm such that it can find a route between any pair of processors if one exists [Tan81]. With this assumption the requirement that the logical network be able to tolerate k failures is equivalent to requiring the underlying physical network provide $k+1$ vertex disjoint paths between all pairs of processors that can reference the object. An abstraction of the underlying network is that it is a graph in which each processor is represented by a vertex and there is an edge between each pair of processors that can communicate directly. We assume the communication is bi–directional so the edges are undirected. We say that an undirected graph is j–connected if j is the minimum number of vertices that have to be removed from it in order to leave more than one connected component behind.

If the graph representing the underlying network is less than j-connected then by definition there is a set of fewer than j vertices that can be removed to leave at least two connected components that cannot communicate with one another. If the logical network is to be k–fault tolerant then clearly the underlying physical network must be at least $(k+1)$–connected. Dolev [Dol81] pointed out that one way of stating Menger's Theorem [Tut84] is that if a graph is at least j–connected then there are at least j vertex disjoint paths betweeen any pair of vertices. For the logical network to be k fault tolerant it is therefore a necessary and sufficient condition that the physical network be at least $k+1$ connected .

It is not necessary that the entire network be $k+1$ connected. What is required however is that k–fault tolerant applications be constrained to sub–networks with this connectivity. Local networks are examples of densely connected sub–networks which provide complete connectivity. Reliable applications on such networks are a possibility. Long–haul networks tend to be less densely connected. For instance, the University of

Washington is connected by only two edges to the rest of ARPANET. Highly available applications that span the less densely connected parts of such a network are therefore impossible.

8.1.2 Reliable Distributed Objects.

It does little good to provide a mechanism guaranteed to find some object in the presence of any k faults if the object itself is not k–fault tolerant. The distributed object must be defined in such a way that it can function in the presence of k failures. This implies that it needs to be distributed among at least $k+1$ processors. If it designates $k+1$ components at which it is willing to receive access requests then we we will guarantee that with up to k failures that an access request message can be delivered to at least one of these components that is still functioning. It is the responsibility of the object's implementation to guarantee that the request can be served under these conditions.

The forwarding address protocols for atomic objects that we presented in the previous chapters could fail to find an object even if the processor at which it is located is in fact functioning. A minimal kind of fault tolerance that we could add is to ensure that we be able to learn the location of the atomic object even if it is down. The protocol could at least reply with a message meaning "Object x is at processor P which is down at the moment". If a processor can truthfully send this reply then it must have a data item that contains the current truth about the object's location. Since this data item must be modified when the object is moved it must be considered part of its representation. This data item is one of the object's *handles*. We have an object distributed among $k+1$ processors, but in order to really access it a particular one of these must be functioning.

Instead of providing only this limited form of fault tolerance, the designer of the object could use a scheme that ensures that the attempt to access the object can be handled in a more meaningful way. Rather than attempt to invent our own strategy for implementing reliable distributed objects, we refer the reader to the several techniques [Gif79, Lam78a, Tho79] that have appeared in the literature. Any of these could be adapted for our purposes. We note that to provide availability in the presence of k faults

these techniques typically distribute the object over more than $k+1$ processors.

One of the major concerns with building reliable distributed entities is that they be kept consistent in the presence of faults. The techniques that have appeared in the literature are particularly concerned with the case in which faults partition the network so that functioning components of the object are left on processors that are unable to communicate with one another. Because our definition of accessibility is that *any* functioning processor be able to access the replicated object we required that k faults not be able to partition the network. In our model all of the functioning components of the distributed object will still be able to find and communicate with one another. It is therefore possible use a weaker scheme for ensuring the object's internal consistency than those cited.

Since our concern is how to find a distributed object reliably rather than how to implement it we will leave the discussion of how to implement a highly available distributed object at this point. We assume that if an object is going to be k–fault tolerant that it will designate a set at least $k+1$ *handles* and that if an "AccessRequest" message can be delivered to a functioning processor with a handle then the object will be able to respond in a reasonable way to it. To find such an object in the presence of k faults requires that we define an forwarding address mechanism that provides at least $k+1$ vertex disjoint forwarding address paths to that set of handles.

8.1.3 Amoebae and Molecules.

With respect to implementing an Object Finding Package (OFP) we distinguish two different ways of implementing reliable distributed objects. In one scheme the distribution is implemented entirely by the Application Programming Environment without modifying the Object Finding Package. In this scheme the components of the distributed objects are atomic objects, each with its own identity. We call such distributed objects *molecules*. While the APE might implement another level of naming for distributed objects at the level of the interface between the APE and the OFP a distributed object is referenced by referencing its component atoms.

In the alternative scheme the components of a distributed object not have their own identities as objects. We call objects implemented this way *amoebae* (after Lamport [Lam78a]) to emphasize a biological analogy. An object of this sort can move by crawling around the system by moving its components around like a biological amoeba extending its pseudo–pods. Furthermore, like a single–celled organism the individual components of these objects would not necessarily be "viable" by themselves. A large enough collection would, however, be able to function and perhaps repair itself.

If reliable distributed objects are implemented as amoebae then the Object Finding Package of Chapter 6 must be extended to support them. Rather than simple forwarding addresses that point to a single processor, it would use the generalized forwarding address mechanism introduced in Chapter 2.

8.2 Finding Molecules: Independent Forwarding Address Update.

The simplest approach to finding molecules is to not modify the Object Finding Package for atomic objects at all. Each processor treats the molecule as a collection of independent atoms and it independently updates its forwarding addresses for each of the molecule's components as it discovers that it has moved.

To ensure k-fault accessibility the molecule must be composed of at least $k+1$ component atoms located at $k+1$ different processors. Each processor that wants to access the molecule needs to keep at least one forwarding address for each component. To ensure that the failure of processors on the forwarding address paths cannot make all of the component atoms inaccessible from a functioning processor the forwarding address paths to them must be disjoint.

Recall that the only processors that appear in the interior of a forwarding address path are the processors at which the object has been located. If any component atom of a distributed object is constrained never to be located at a processor that another component has visited then their forwarding address paths will be forever disjoint. The APE can enforce this condition when an object proposes to move to a processor by

checking the 'seenHere' flags at that processor of all the molecule's component atoms.

This constraint on the object's ability to move is the result of allowing the OFP to continue to update each component atom's forwarding address in a separate atomic action. Rather than explore the implications of the strategy of forcing a processor to update all forwarding addresses for a molecule simultaneously we jump directly to the issue of finding an amoeba using generalized forwarding addresses. The same strategy could be simulated by the APE at a higher level.

8.3 Finding Amoebae: Generalized Fowarding Addresses.

An amoeba that is k–fault accessible must have at least $k+1$ handles through which it can be accessed. To find these handles we use generalized forwarding addresses as defined in Chapter 2. For each object that it can name a processor keeps a single forwarding address defined as follows:

```
FA = struct[name: CID,
            loc: sequence[ProcID],
            ts: TimeStamp].
```

The extension is that instead of being located at a single processor the location of an amoeba is defined to be the collection of processors at which its handles are simultaneously located. Some of the handles might play special roles with respect to the object so we allow them to be distinguished from one another by the places they occupy in 'loc'. When one or more handles are moved then the new location of the amoeba is the new collection of processors. Even though the new location may contain many of the processors in the old location they should be thought of as distinct entities. A new forwarding address is created whenever an amoeba moves any of its handles and all processors participating in the move *must* store a copy of that new address. This includes all processors from which handles were moved plus all of the processors in the new location. As in the basic forwarding address mechanism any other processor may discard a forwarding address only by replacing it with a newer one for the same

object. These two conditions ensure that the graph of forwarding addresses for the object remains connected and that any processor identifier within a forwarding address points to a processor that is in the object's current location or that has a more recent forwarding address. This implies that there are no cycles in an object's forwarding address graph that are not contained entirely within its current location.

The various operations on forwarding addresses defined in Chapters 6 and 7 can be extended in the obvious way to use generalized forwarding addresses containing $k+1$ processor identifiers. We claim that such an extension results in k–fault accessibility. This is restated in the following theorem.

Theorem 13 *Let x be an amoeba and let P be a processor with a (generalized) forwarding address for x. If k processors other than P fail then for there to be an unaffected forwarding address path to a non–failed processor that holds one of x's handles it is necessary and sufficient that x always have $k+1$ handles on distinct processors and that the* 'loc' *field in all forwarding addresses for x point to such a configuration.*

Proof: The case in which P holds one of x's handles is special. P's forwarding address for x points to itself and P does not fail by assumption.

If P does not hold one of x's handles then its forwarding address does not contain a pointer to P. If x's handles are on fewer that $k+1$ processors and if they all fail then no processor can find a handle on a non–failed processor. Similarly if P has a forwarding address for x that points to fewer than $k+1$ distinct processors then if these all fail then P is cut off from x's handles. This establishes necessity.

We now show that $k+1$ handles and pointers in forwarding addresses are sufficient by showing how to construct the required path. Let Q be any non–failed processor with a forwarding address for x. As above, if Q holds one of x's handles then the forwarding address points to Q. By assumption, Q does not fail. If Q does not hold one of x's handles then because there are $k+1$ pointers in the forwarding address but only k failed processors at least one of the processors to which Q's forwarding address points must also not fail. Call this processor Q'. The rules for moving amoebae and for updating

forwarding address tables imply that Q' either holds a handle for x or has a more recent forwarding address for it than Q. In other words, if Q is not the holder of one of x's handles then there is an extension of every forwarding path for x through Q to a non–failed processor. Because the time stamps must increase monotonically Q' is not already in the path. By transitivity there is thus a forwarding address path involving only non–failed processors from any non–failed processor P to a non–failed processor in the current location of x. □

The proof suggests a possible implementation of a protocol to remotely access an amoeba. Each processor along the forwarding address path need only be sure that it passes "AccessRequest" messages to non–failed processors. It is still possible that a processor will fail after receiving an "AccessRequest" and before passing it on, so care must be used in designing the low–level protocol so that there are appropriate acknowledgements and a reasonable retry strategy. Because we normally expect the system to be reliable a reasonable scheme might be to first just forward the "AccessRequest" with minimal handshaking to ensure that at each step it has been sent to a non–failed processor. If this has not succeeded after a reasonable time then the originator of the request can try to track down the object using a sequence of "WhereIs" messages at each step. We leave the detailed design of such a protocol as an exercise.

The statement of the theorem is in terms of the static structure of the forwarding address graph. If x moves while the "AccessRequest" message is being forwarded then it may move to a processor that the message has already passed. This has the effect of replacing the forwarding address at that processor so the new forwarding address graph will remain acyclic except among the holders of x's handles. The time stamps of forwarding addresses along the path taken by the message is guaranteed to increase monotonically. Thus, the message will eventually catch up with x as long as it is forwarded faster than x moves.

8.4 Reconfigurability.

A stronger notion than accessibility is reconfigurability. When a disaster occurs a processor together with its stable storage becomes permanently unavailable. It is desirable that the system be able to reconfigure itself after a disaster. Within the context of this discussion a reconfiguration has two components. A distributed object that has been affected by a crash or disaster must be repairable by restoring lost redundancy so that it will be able to tolerate further failures. Similarly, the object finding mechanism must also be reconfigured to replace the data items lost on the destroyed processors.

Lamport [Lam78a] discusses the issue of the reconfiguration of an amoeba and presented a set of conditions under which it is possible.

The inexact reference sets defined in Chapter 7 can be added to the generalized forwarding addresses introduced in this chapter. All non–failed processors will have reference sets that include all non–failed processors that have forwarding addresses that point to them. Since the processor identifiers represent back pointers along the edges of the forwarding address graph a corollary of Theorem 13 is that the non–failed processor(s) holding x's handles will be able to find all non–failed processors that can reference x and inform them of the nature of the reconfiguration. Care must be used with a processor that has crashed rather than been destroyed. The reconfiguration scheme may optionally help it to recover from its crash, destroy it so that it never recovers, or allow it to recover on its own if it can be determined that it will be safe to do so.

Note that there is no advantage to trying to use exact reference sets. In the presence of crashes and disasters they will become inexact anyway. Such sensitivity to failures is a major weakness of the reference count technique. General forms of garbage collection exhibit similar problems.

8.5 Other Aspects of Distributed Objects.

In addition to being used to enhance reliability distributed objects can also be used to increase efficiency. Replicated items can allow more concurrent access than atomic

objects. For instance, Gifford's weighted voting scheme [Gif79] can be used to tune the object for both efficient concurrent access and enhanced availability.

Another way that distribution can be used to increase efficiency is to improve the cost of finding an object using forwarding addresses. If an object is distributed by giving it a handle that never moves, then the cost of finding it through that handle will be constant. This is essentially what happens when an object has an entry in a directory that is required to have the truth about its location. Because the directory entry must be updated every time the object is moved it is part of the representation. This is one way of implementing such directories using a forwarding address mechanism.

Chapter 9

Recapitulation and Conclusion.

In the first two chapters we introduced an *object finding problem.* It can be viewed as a restricted case of the *name resolution problem.* One way that it is differentiated from other forms of name resolution is that the *objects* that are to be found are analagous to physical entities. These objects can move from processor to processor in a sequence of discrete events that simulate the continuous motion of physical objects. Records of these events can witness the path of the object on its travels. Another way that our object finding problem is different from other name resolution problems is that it uses only *proper names* that are defined in such a way that there is a chain of events that leads back from every instance of such a name to the object it references. Records of this chain of events witness the validity of the name and can be used to help find the object given the instance of the name.

Within this model of the object finding problem we identified the use of *forwarding addresses* as an obvious and natural mechanism for finding objects in a large distributed computer system. The forwarding address mechanism has the advantages that it is completely decentralized, it allows a set of processors cooperating in an application to autonomously go about their business without interference or help from outside, and, if used judiciously, can be as efficient as any other mechanism for finding objects.

When used in an arbitrary way some individual operations involved in the use of forwarding addresses can be made expensive if the object being sought visits a large number of processors. In Chapter 3 we performed analyses of the costs of using some

simple forwarding address protocols. These analyses show that the average case cost of using forwarding addresses is reasonable. If path compression is used then the ammortized cost of accessing an object is $\Theta(\log N)$, where N is the number of processors the object visits in its lifetime. Note that for all protocols we analyzed the cost of finding an object does not depend upon the total size of the system, rather on the number of processors that can name that object.

The analyses of these protocols are interesting in their own right. The average case analyses of two of the protocols are done by modeling the state of the forwarding address tree as a Markov chain. Under the assumption that the destination of an object move is taken from a uniform random distribution, one of these protocols generates random labeled trees. Our analysis of it improves upon the previously published results for the probability distribution of the depth of the vertices in such a tree. The worst case analysis of the protocol that uses path compression also tightens the best published analysis of a sub-optimal algorithm for the set union problem [TvL84].

There are other issues besides cost that need to be considered in evaluating whether a mechanism is suitable for use in a real system. These include the need for reasonable storage management, reliability, and a degree of administrative control. To explore how these issues can be dealt with using forwarding addresses we presented pseudo–code implementations of some model object finding services that use forwarding addresses.

We presented the first of these in Chapter 6. This was a model implementation for atomic objects in a system that has stable transactional storage. One motivation for presenting this was to elaborate on the model of proper names and objects that we presented in Chapter 1. Of particular interest from this standpoint is the interrelationship between object identity and the naming system that is revealed by our suggested implementation of the move operation for objects.

Another motivation for presenting the model implementation of this basic forwarding address mechanism was to fill in much of the detail missing from the simple models of forwarding address protocols used in Chapter 3.

One possible objection to the use of forwarding addresses is the problem with the

management of storage occupied by forwarding addresses. Chapter 7 deals with storage management issues. The *inexact reference set* extension is the major contribution of this chapter. It explicitly represents the contractual obligations to which each processor may have committed itself. Using this mechanism a processor can not only discover which obligations do in fact exist, but it can re–negotiate those obligations. This ensures that not only can storage be recovered when an obligation is removed, but also that the system can be reconfigured under administrative control.

Because inexact reference sets have a weaker consistency condition than reference counts they can be cheaper to use than reference counts. In particular they do not require the use of multiple–processor transactions. Another advantage of the weak consistency condition is is that inexact reference sets can be used in the presence of processor failures. This is not the case for reference counts.

Global garbage collection on large distributed systems is a difficult and unsolved problem. This is because if any processor in a system can refer to an object then all processors must participate in garbage collection. This can be very expensive and unreliable. By using inexact reference sets for objects referenced by proper names a processor can efficiently learn which other processors might be able to refer to an object whose representation it has. This information can be used to localize the attempts to reclaim storage.

Another possible objection to the use of forwarding addresses is that the failure of a processor along a path of forwarding addresses might prevent a funtioning processor from finding a functioning object. Most of the time this will be no problem. On the other hand there will be applications that demand continuing accessibility, even in the presence of faults that not only halt the processor but destroy all data stored there. In Chapter 8 we deal with this issue. While it is possible to implement distribution for reliability at a higher level, we present an implementation based on a generalization of forwarding addresses to include not just a single processor identifier, but the identifiers of the collection of processors simultaneously occupied by the distributed object. This

generalization guarantees the accessibility of such a distributed object from any non-failed processor and it does not restrict the motion of the object.

9.1 Open Problems

While we have argued that forwarding addresses are an excellent mechanism for use in a very large decentralized system to find mobile objects given causally connected proper names for them, no such system yet exists. We do not know whether the paradigm used in the fictitious Humongous Corporation's "Humongous Object Programming System" would in fact be a viable mechanism for creating a very large system that contains many autonomous distributed applications. We do know, however, that without a good method for finding objects that such systems cannot be successful. Experimental systems such as Eden [ABLN83] are a step in the right direction, but they are not yet large enough nor are their objects small and mobile enough to use this paradigm. We would like to see further experiments in which objects are made more mobile and we believe that forwarding addresses would be appropriately used there.

The interaction between routing and object finding can use more investigation. Because of the sheer number and small size of the objects in our postulated system we do not believe that it is appropriate for the underlying network to know about them and to route messages to them. This is in contrast to Halstead's proposed system [Hal79] in which all of the physical routing information is stored in the tree of forwarding addresses.

Some of the techniques presented in Chapters 7 and 8 might be used in reliable adaptive routing algorithms.

There is an open question as to whether there is a more efficient decentralized algorithm for finding objects in the abstract model we use for our analyses. As we pointed out in Chapter 3 the dispatching of a path compression when a move is performed does not result in a smaller lower bound. In contrast, if we allow the protocol to use one of various forms of what we called *ac hoc* centralization then object finding can be done in constant time.

Mullender and Vitányi [MV85] used a different model of naming and a different notion

of decentralization in their analyses. We leave as completely open the issues of trying to systemetize and unify these disparate models of naming, objects, and of decentralization.

There are, of course, the various open problems that were brought up in the analyses of costs. In particular, we still have no analytic solutions to the average case analysis of the *Jacc* protocol.

Bibliography

[ABLN83] Guy Almes, Andrew Black, Edward Lazowska, and Jerre Noe. *The Eden System: A Technical Review.* Technical Report 83-10-05, University of Washington, October 1983.

[BHJL85] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. *Distribution and Abstract Types in Emerald.* Technical Report TR 85–08–05, Department of Computer Science, University of Washington, August 1985.

[BLNS82] A. D. Birrell, R. Levin, R.M. Needham, and M.D. Schroeder. Grapevine: an exercise in distributed computing. *Communications of the ACM*, 25(4):260–274, April 1982.

[Bog82] David R. Boggs. *Internet Broadcasting.* PhD thesis, Stanford University, January 1982.

[Car56] Rudolph Carnap. *Meaning and Necessity, edition 2.* University of Chicago Press, Chicago, 1956.

[CM84] David R. Cheriton and Timothy P. Mann. Uniform access to distributed name interpretation in the V-system. In *Proceedings Fourth International Conference on Distributed Computing Systems*, San Francisco, California, May 1984.

[CP85] Douglas E. Comer and Larry L. Peterson. *A Name Resolution Model for Distributed Systems.* Tilde Report CSD-TR-99, Purdue, February 1985.

[Dan82] Roger Berry Dannenberg. *Resource Sharing in a Network of Personal Computers.* PhD thesis, Carnegie–Mellon University, December 1982.

[DH85] Roger B. Dannenberg and Peter G. Hibbard. A butler process for resource sharing on spice machines. *ACM Transactions on Office Information Systems*, 3(3):234–248, July 1985.

[Dio80] J. Dion. The Cambridge file server. *Operating Systems Review*, 14(4):26–35, October 1980.

[DMN70] O-J. Dahl, B. Myhrhaug, and K. Nygard. *The SIMULA 67 Common Base Langauge.* Publication S-22, Norwegian Computing Center, Oslo, Norway, 1970.

[Dol81] Danny Dolev. Unanimity in an unknown and unreliable environment. In *Proceedings of the 22nd IEEE Symposium on the Foundations of Computer Science*, pages 159–168, IEEE, 1981.

[Don72] K. Donnellan. *Proper Names and Identifying Descriptions*, pages 356–379. Volume 40 of *Semantics of Natural Language*, Reidel, Dordrecht, Holland, 2 edition, 1972.

[DR76] Jon Doyle and Ronald L. Rivest. Linear expected time of a simple union–find algorithm. *Information Processing Letters*, 5(5):146–148, November 1976.

[Fel68] William Feller. *An Introduction to Probability Theory and Its Applications.* Volume 1, Wiley, New York, third edition, 1968.

[Fis72] Michael J. Fischer. *Efficiency of Equivalence Algorithms*, pages 153–168. Plenum Press, New York, 1972.

[FM77] Michael J. Fischer and Albert R. Meyer. Course Notes for CSCI 521, University of Washington. 1977. Unpublished.

[Gif79] David K. Gifford. Weighted voting for replicated data. In *Proceedings Seventh Symposium on Operating Systems Principles*, pages 66–81, ACM, December 1979.

[Gif82] David K. Gifford. *Information Storage in a Decentralized Computer System.* Technical Report CSL-81-8, Computer Science Laboratory, Xerox PARC, March 1982.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison–Wesley, Reading, Masachusetts, 1983.

[Gra79] J. N. Gray. Notes on data base operating systems. In R. Bayer, R. M. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, pages 393–481, Springer-Verlag, 1979.

[Hal79] Robert Halstead. *Reference Tree Networks: Virtual Machine and Implementation.* PhD thesis, MIT, July 1979.

[HB77] Carl Hewitt and Henry Baker. Actors and continuous functionals. In *IFIP Working Conference on Formal Description of Programming Concepts*, pages 16.1–16.21, IFIP, St. Andrews, New Brunswick, Canada, August 1977.

[HBS73] Carl Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI-73*, Stanford, Calif., August 1973.

[Hut85] Norman C. Hutchinson. *The Emerald Programming Language Definition.* Oz Project Technical Note, Dept. of Computer Science, University of Washington, Seattle, Washington, June 1985.

[Kat82] Randy H. Katz. A database approach for managing VLSI design data. In *Proceedings of the 19th ACM/IEEE Design Automation Conference*, ACM/IEEE, Las Vegas, Nevada, June 1982.

[Knu68] Donald Knuth. *The Art of Computer Programming, Volume 1/ Fundamental Algorithms.* Addison-Wesley, Reading, Massachusetts, 1968.

[Kri72] Saul A. Kripke. *Naming and Necessity.* Harvard University Press, Cambridge, Mass., 1972.

[LAB*81] Barbara Liskov, Russ Atkinson, Toby Bloom, Eliot Moss, Craig Schaffert, Bob Scheifler, and Alan Snyder. *CLU Reference Manual.* Volume 114 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, 1981.

[Lam78a] Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2, 1978.

[Lam78b] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of the Assoc. for Computing Machinery*, 21(7):558–565, July 1978.

[Lam81] Butler W. Lampson. Atomic transactions. In *Distributed Systems — Architecture and Implementation: an Advanced Course*, chapter 11, pages 246–265, Springer-Verlag, 1981.

[Lam85] Butler W. Lampson. *Fourth ACM Symposium on the Principles of Distributed Computing*, Minacki, Ontario, August 1985. Invited talk; not in the conference proceedings.

[Lev84] Henry M. Levy. *Capability-Based Computer Systems.* Digital Press, Bedford, Mass., 1984.

[Lis84] Barbara Liskov. *Overview of the Argus Language and System.* Programming Methodology Group Memo 40, MIT Laboratory for Computer Science, February 1984.

[LLA*81] Edward D. Lazowska, Henry M. Levy, Guy Almes, Michael J. Fischer, Robert J. Fowler, and Stephan C. Vestal. The architecture of the Eden system. In *Proceedings of the Eighth Symposium on Operating Systems Principles*, Asilomar, California, December 1981.

[LSHL82] P.J. Leach, B.L. Stumpf, J. A. Hamilton, and P.H. Levine. Uids as internal names in a distributed file system. In *Proceedings: Symposium on Principles of Distributed Computing*, pages 34–41, ACM, Ottawa, August 1982.

[MD81] J.G. Mitchell and J. Dion. A comparison of two network-based file servers. In *Proceedings of the Eighth Symposium on Operating Systems Principles*, pages 45–46, ACM, Pacific Grove, California, December 1981.

[MM70] A. Meir and J.W. Moon. The distance between points in random trees. *Journal of Combinatorial Theory*, 8:99–103, 1970.

[MM78] A. Meir and J.W. Moon. On the altitude of nodes in random trees. *Can. J. Math.*, 30(5):997–1015, May 1978.

[Mor73] James H. Morris, Jr. Types are not sets. In *ACM Symposium on Programming Languages*, pages 120–124, ACM, Boston, 1973.

[Mos81] J. E. B. Moss. *Nested Transactions. An Approach to Reliable Distributed Computing.* Technical Report MIT/LCS/TR-260, MIT Laboratory for Computer Science, 1981.

[MV85] Sape J. Mullender and Paul M. B. Vitányi. Distributed match-making for processes in computer networks. In *Proceedings Fourth ACM Symposium on the Principles of Distributed Computation*, ACM, Minacki, Ontario, August 1985.

[OD83] D.C Oppen and Y.K. Dalal. The Clearinghouse: a de- centralized agent for locating named objects in a dis- tributed environment. *ACM Tran. on Office Information Systems*, 1(3):230–253, July 1983.

[Par67] Emanuel Parzan. *Stochastic Processes.* Holden-Day, San Francisco, 1967.

[PM83] M.L. Powell and B.P. Miller. Process migration in DEMOS/MP. In *Proc. 9th ACM Symposium on Operating Systems Principles*, pages 110–119, ACM, October 1983.

[RS67] A. Rényi and G. Szekeres. On the height of trees. *The Journal of the Australian Mathematical Society*, 7, part 4:497–507, November 1967.

[Sal78] Jerome H. Saltzer. *Naming and Binding of Objects*, pages 99–208. Volume 60 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, 1978.

[San85] Jan Sanislo. 1985. personal communication.

[SBN84] Michael D. Schroeder, Andrew D. Birrell, and Roger M. Needham. Experience with Grapevine: the growth of a distributed system. *ACM Transactions on Computer Systems*, 2(1):3–23, February 1984.

[Sch82] Eric Emerson Schmidt. *Controlling Large System Development in a Distributed Environment.* PhD thesis, University of California, Berkeley, 1982. Also XEROX PARC CSL82-7.

[Sco70] Dana Scott. *Advice on Modal Logic*, pages 143–179. D. Reidel, Dordrecht-Holland, 1970.

[SMI80] H.E. Sturgis, J.G. Mitchell, , and J. Israel. Issues in the design and use of a distributed file system. *Operating Systems Review*, 14(3):55–69, July 1980.

[Sol85] Karen Rosin Sollins. *Distributed Name Management.* PhD thesis, MIT, Department of Electrical Engineering and Computer Science, February 1985.

[SRC84] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[SS81] Dale Skeen and Michael Stonebraker. A formal model of crash recovery in a distributed system. In *Proceedings of the Sixth Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 129–142, Lawrence Berkeley Laboratory, Berkeley, 1981.

[Str59] Peter Strawson. *Individuals: An Essay in Descriptive Metaphysics.* Methuen, London, 1959.

[Tan81] Andrew S. Tannenbaum. *Computer Networks.* Prentice Hall, Englewood Cliffs, New Jersey, 1981.

[Ter85] Douglas Brian Terry. *Distributed Name Servers: Naming and Caching in Large Deistributed Computing Environments.* PhD thesis, University of California at Berkeley, 1985.

[Tho79] R.H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.

[Tut84] W. T. Tutte. *Graph Theory.* Volume 21 of *Encyclopedia of Mathematics and Its Applications*, Addison–Wesley, Reading, Massachusetts, 1984.

[TvL84] R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *JACM*, 31(2):245–281, April 1984.

[Wat81] Richard W. Watson. *Identifiers (Naming) in Distributed Systems*, pages 191–210. Volume 105 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, 1981.

[Yao85] Andrew C. Yao. On the expected performance of path compression algorithms. *SIAM Journal on Computing*, 14(1):129–133, February 1985.

BIBLIOGRAPHICAL NOTE

Robert Joseph Fowler was born March 9, 1949 in Hartford, Connecticut. He graduated from Bulkeley High School in Hartford in 1967. In 1971 he received the A.B. degree (*cum laude* in physics) from Harvard College. He received an M.S. in Computer Science from the University of Washington in 1981.