Productivity Engineering in the UNIX[1] Environment

Distributed Name Servers: Naming and Caching in Large Distributed Computing Environments

Technical Report

S. L. Graham
Principal Investigator

(415) 642-2059

"The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government."

Contract No. N00039-84-C-0089

~~August 7, 1984 - August 6, 1987~~

Dec 1985

Arpa Order No. 4871

[1] UNIX is a trademark of AT&T Bell Laboratories

Distributed Name Servers:
Naming and Caching in Large Distributed Computing Environments

by

Douglas Brian Terry

Abstract

Name services facilitate sharing in distributed environments by allowing objects to be named unambiguously and maintaining a set of application-defined attributes for each named object. Existing distributed name services, which manage names based on their syntactic structure, may lack the flexibility needed by large, diverse, and evolving computing communities. A new approach, structure-free name management, separates three activities: choosing names, selecting the storage sites for object attributes, and resolving an object's name to its attributes. Administrative entities apportion the responsibility for managing various names, while the name service's information needed to locate an object's attributes can be independently reconfigured to improve performance or meet changing demands.

An analytical performance model for distributed name services provides assessments of the effect of various design and configuration choices on the cost of name service operations. Measurements of Xerox's Grapevine registration service are used as inputs to the model to demonstrate the benefits of replicating an object's attributes to coincide with sizeable localities of interest. Additional performance benefits result from clients' acquiring local caches of name service data treated as hints. A cache management strategy that maintains a minimum level of cache accuracy is shown to be more effective than the usual technique of maximizing the hit ratio; cache managers can guarantee reduced overall response times, even though clients must occasionally recover from outdated cache data.

Distributed Name Servers:
Naming and Caching in Large Distribu:ed Computing Environments

by

Douglas Brian Terry

Contents

List of Figures

List of Tables

Acknowledgements

The journey in pursuit of a doctoral degree is long and perilous; no one can make it alone. I am grateful to the many people that have aided me along the way.

My two major advisors, Robert Fabry and Domenico Ferrari, gave me the freedom to explore on my own, while fixing an eye on my wanderings. I learned a tremendous amount from my explorations. Bob's insistence on excellence served to reinforce my own. His suggestions for improvements, right until the end, substantially strengthened the thesis. Domenico, in spite of his many responsibilities and unreasonable work load, would always find time to speak with me whenever I needed guidance. With his red pen in hand, he thoroughly and punctually marked up every draft. My third reader, Lucien LeCam, provided a much needed refresher course on probability and statistics.

I appreciate the support of several colleagues that have contributed in various ways. Bob Hagmann was instrumental in arranging a consulting agreement with Xerox PARC so that I could study the Grapevine system. Michael Schroeder and Andrew Birrell explained the internals of Grapevine and how its logs were organized. Hal Murray was a constant source of information on the day-to-day operation of the system. More importantly, Hal read the dissertation and discovered several embarrassing bugs in my name server prototype implementation. Howard Sturgis bravely read an early draft and helped me focus my ideas. Luis Felipe Cabrera, Juliet Sutherland, and Songnian Zhou furnished comments on several chapters. Juliet was particularly helpful at keeping me abreast of current standardization efforts in the area of name services. I have also benefited from many discussions with fellow researchers at U. C. Berkeley, IBM Research, Xerox PARC, and other institutions across the country too numerous to name.

My interest in distributed computing and the motivation for my thesis germinated while I was an academic associate for IBM Research. This research was partially sponsored by the Defense Advance Research Projects Agency (DoD) Arpa Order No. 4031 and monitored by the Naval Electronic System Command under Contract No. N00039-C-0235. Much of the support during the preparation of this dissertation was generously provided by the Xerox Palo Alto Research Center.

Margaret Butler played many vital roles: technical editor, style critic, counselor, friend. She patiently suffered through numerous rough drafts. No matter what predicament I got myself into, she was always just a phone call away. Margaret invariably had a smile and good word when I was down. I am deeply indebted to her for all of her assistance. I only hope that I can serve her as well when the roles are reversed.

As usual, my friends and family have kept me going through all these years. I am eternally grateful for all the joyous moments I have had in Berkeley. Katie deserves special credit for helping me adjust to the strangeness that is Berkeley and to the demands of graduate studies.

Finally, I dedicate this dissertation, the culmination of many years of education, to my parents, George and Georgette Terry. They may not understand its technical merits, but their contributions have been great.

Chapter 1

Introduction

> *I knew very well that in addition to the great planets – such as the Earth, Jupiter, Mars, Venus – to which we have given names, there are also hundreds of others, some of which are so small that one has a hard time seeing them through the telescope. When an astronomer discovers one of these he does not give it a name, but only a number.*
>
> — Antoine de Saint Exupéry, *The Little Prince.*

1.1 The Electronic Baobabs

Like the little prince's galaxy, with planets too numerous to be named, contemporary distributed computing environments have evolved to the point that it is difficult to name and catalogue the many available resources. To facilitate the sharing of information and resources, immense interconnections of public and private data networks have been established, permitting users access to extraordinary numbers of potentially shareable resources. The DARPA Internet hosts table, for instance, now contains over 300 networks connecting most of the major U.S. universities, military organizations, and computer corporations.

Physical connectivity, however, is not sufficient to allow resources to be effectively utilized by the various members of these vast, interconnected computing communities. Uniform mechanisms are needed for *identifying* and *locating* objects and resources that are made accessible to the community by their creators or owners. That is, objects should be given *names*, names that can be freely passed around the internet and shared amongst its users so that the objects themselves might be shared. Once users have a way of referring to objects, services should be provided for locating particular objects and discovering how to access those objects.

This dissertation addresses the issues of providing such a *name service*[1] for a widely distributed computing environment. It progresses in three stages: First, general techniques for managing names in a distributed manner are developed. Second, the performance of such techniques for large name services is analytically modeled. Third, a client's level of performance is enhanced by introducing caches of naming data. The next section discusses the nature of name services, providing the background for the remainder of the dissertation.

[1] Throughout this dissertation, terms appear in italics when they are first introduced; their definitions are reproduced in the glossary for later reference.

1.2 Name Services

1.2.1 Role

A *name service* enables its clients to name resources or objects and provides facilities for accessing information about these objects. The term *object* will be used hereafter to refer to anything that deserves a name. Both physical and logical entities may be objects. For instance, computers, file servers, printers, disk drives and files can all be objects. Processes, services, distribution lists, and computer messages can also be objects, as may computer programmers, operators, and technicians. Some objects exist within the bounds of the distributed computer system, while others have a life of their own. Note also that some objects are *active*, such as a process executing a program, while others play a *passive* role, and hence must be acted upon and managed by active objects.

The *client/server* model of distributed computing has become popular in describing relationships between active objects. *Servers* offer services to *clients* that may make use of those services. Often, an active object is both a provider of some services and a client of others. Since servers and clients may exist at various locations in a distributed computing environment, means must be provided for establishing liaisons between them.

The name service is a "master" service, which acts as a rendezvous point for other servers and clients of the services provided by those servers. Services can be made available to the general community by *registering* them with the name service. The information presented on the "registration form" includes the name of the service and information needed to make use of the service. A client of a service may obtain this information by contacting the name service and presenting the name of the desired service. From that point on, the client and server may establish a direct connection to conduct their business.

The name service thus enables other services to be identified and accessed in a uniform way [Abraham and Dalal 80]. Members of a large distributed computing community need only know how to access the name service in order to gain access to a multitude of services indirectly through the "well-known" name service. For passive objects, the name service maintains information that allows them to be manipulated and shared by specialized services.

A name service, described here as a general name management facility, provides more than the usual name-to-address bindings. It subsumes services such as directory systems for electronic mail, file name managers, and database catalog managers. These services can be viewed as name services specialized for a particular application domain; for example, the catalog manager for a distributed database management system maintains information about named database objects, such as their locations, access controls, and statistics used for query optimization [Martella and Schreiber 80].

Much of the current confusion and difficulty in interconnecting existing distributed environments seems from the fact that various incompatible name services are being employed for widely-used applications, like mail. Much can be gained from adopting uniform name services. However, the question of whether a single general name service should be used for all objects or whether specialized name services should continue to exist with a global name service used to locate the more specific services is difficult to answer. The choice is not a critical one to the discussions that follow, as the issues remain the same.

1.2.2 Names

1.2.2.1 Properties

Simplistically, a *name* is a character string that identifies an object. However, there is a general lack of consensus about what properties distinguish names from other types of identifiers. John Shoch made the following incisive, albeit vague, distinction between three types of identifiers used in computer networks [Shoch 78]:

> "The *name* of a resource indicates what we seek,
> an *address* indicates where it is, and
> a *route* tells us how to get there."

Jerome Saltzer, on the other hand, suggests a broader use of the word "name" and portrays the relationship between names and addresses as bindings; that is, he defines an object's "address" to be a "name of the object it is bound to" [Saltzer 82].

This dissertation draws a simple distinction between names and addresses: *names are chosen by users, whereas addresses are assigned by the system or system administrators.* This distinction complies with Shoch's basic terminology and resembles Richard Watson's distinction between human-oriented names and machine-oriented identifiers [Watson 81]. Historically, the use of names in communication networks emerged as a convenience to humans, who find it difficult to remember numbers denoting the addresses of network entities. Names, as characterized herein. may be:

- readable by humans and of mnemonic value,
- independent of network locations.

The first property arises naturally since humans tend to choose names that describe their referents [Carroll 78]. The second property allows an object to migrate to a new location in the distributed environment without changing its name, and hence without requiring changes in others' references to the named object[2].

The interpretation of names presents additional properties: A name is *unambiguous* if and only if it refers to at most one object. That is, the same name cannot be used by different clients of the name service to refer to different objects. A name is *unique* if it represents the *only* name for its referent. Several *non-unique* names may identify the same object. Often, in such cases, one name is recognized as the preferred name and the others are called *aliases* or *nicknames*. Note that some people use the terms "unique" and "unambiguous" interchangeably. As defined here, ambiguity corresponds to a one-to-many relationship between names and objects, whereas non-uniqueness suggests a many-to-one binding.

A name is said to be *global* or *absolute* if it is interpreted in a consistent manner by all clients and services, regardless of their location in the environment or other factors. Absolute names may be freely passed around from object to object without affecting their interpretation. On the other hand, *relative* names are interpreted according to some state information.

The name services of interest in this dissertation manage unambiguous names so that dialogues for resolving ambiguities are not required. In addition, they can guarantee the uniqueness or absoluteness of names, but the general mechanisms do not assume that these properties are always desired by applications making use of a name service.

1.2.2.2 Structure

The convention adopted for naming objects dictates the syntactic representation of names, as well as their semantic interpretation. The set of names complying with a given *naming convention* is called the *name space*.

Names are commonly structured as a series of alphanumeric *labels* interleaved with various *separation characters*. Although many separation characters are in common use in existing naming conventions, including '@', '%', ':', '.', '/', and '!', the '.' will be used for simplicity hereinafter, except in cases where a specific naming convention is being discussed. Thus, the name "A.B.C" consists of three labels, "A", "B", and "C".

[2] Addresses may be location-independent as well; these are occasionally referred to as *logical addresses* [Rosen 81]. Some recent proposals purporting new approaches to name management are really suggestions for managing logical addresses in the communication transport layer [Cheng and Liu 82] [Cheng 84] [Chesley and Rom 83].

A *component* of a name is a substring of that name composed of one or more labels and the embedded separation characters. The name "A.B.C" contains the following components: "A", "B", "C", "A.B", "B.C", and "A.B.C".

Abbreviations are short forms for names that may be used in certain circumstances as a substitute for the complete name. An abbreviation differs from an alias in that it is a component of a name, that is, syntactically derived from the name, and is not treated as a fully qualified name. As such, abbreviations are not generally recognized by the name service. Usually, abbreviations are provided by an application as a convenience to human users, who do not like to type long names, and converted in an application-specific way to a fully qualified name before being presented to the name service. As an example, consider a mail system that names mail recipients according to the convention "user.host"; the system may choose to accept a name of the form "user" as an abbreviation for "user.this-host".

1.2.2.3 Contexts

Names always exist within some context. A *context* can be loosely defined as the environment in which a name is valid. In many programming languages, the notion of a context is instantiated as the *scope* of a variable. In distributed systems, contexts represent a partitioning of the name space, often along natural geographical or organizational boundaries. A name may naturally occur in more than one context, and contexts may be nested. For instance, the login name "terry" exists within the contexts of both "Berkeley" and "Xerox". In turn, "Berkeley" exists within the context of the "University of California", which exists within the context of all universities.

A component of a name may denote a context in which other parts of the name exist. Such a context is called an *explicit context* since it is explicitly represented in the structure of the name. For example, given the name "A.B.C", "B.C" might be viewed as a name existing explicitly in the context of "A".

On the other hand, a context that is not an explicit part of the name is called an *implicit context*. Relative naming conventions involve interpreting a name according to some implicit context. Only if implicit contexts are universal can absolute naming conventions be attained. The name service itself may be one example of a global implicit context.

The "dot" notation used for delineating the labels of a name does not contain enough information to indicate the contextual interpretation of the name. For one thing, some naming conventions may choose to nest contexts left-to-right while others use right-to-left associativity. Moreover, not all of the labels of a name necessarily represent contexts. In this dissertation, a name will be presented in the form "context(subname)" when the contextual structure of the name is important. For example, the name "A.B.C" could be expressed as "A(B.C)", indicating that the subname "B.C" should be interpreted in the context of "A". Alternatively, "C" could exist in the context of "A.B", or "A.B" could exist in the context of "C"; these would be written as "A.B(C)" and "C(A.B)", respectively. If the three labels were nested contexts, the name might be "A(B(C))".

With explicit contexts, a sufficient condition for achieving unambiguous names can be recursively given as follows: *the name "context(subname)" is globally unambiguous if the subname is unambiguous within the context, and the context has a globally unambiguous name.*

1.2.3 Object attributes

The information maintained about a named object by the name service consists of a set of *attributes* for the object. Object attributes have both a *type* and a *value*, where the type indicates the format and meaning of the value field. The name service does not attempt to interpret an attribute value. Thus, applications making use of the name service must agree on the structure and semantics associated with object attributes. Agreeing on the format of attribute values is particularly important in a heterogeneous environment where machines have different word sizes, number representations, bit orientations, and so on.

Names that have a list of names as an attribute, generically called *group names*, are used for such things as mail distribution lists and access control lists. One way of representing these membership lists is with a single attribute of type "MembersAre" that takes a list of names, perhaps separated by commas, as a value. Alternatively, each member could be listed as a separate attribute of type "HasMember". The first representation makes it easy to enumerate the membership set, while the second is more convenient for adding and removing individual members. This illustrates the amount of freedom available in choosing various attributes and their representations.

Generally, the types of attributes for an object vary with the type of the object. For instance, information about a user, including anything from his office phone number to his address for receiving electronic mail [Feinler 77], differs radically from information about files [Mogul 84] [Leach *et al.* 82] or database objects, such as the data's location, structure, availability, and usage [Allen *et al.* 82] [Martella and Schreiber 80] [Lindsay 80]. The name service may choose to restrict the types of attributes or require certain attributes for given classes of objects [Cooper 82].

In a layered system, such as the Open Systems Interconnection reference model adopted by ISO [ISO 81], an attribute for an object often represents an identifier to be presented to the next lower layer. The binding of names to network addresses, which motivated the conception of name services, represents a good example of this. For communicating with a object, one might need an attribute for the object of type "InternetAddress", whose value is a communication socket particular to the communication protocol being employed. Using the DARPA Internet Protocol [Postel *et al.* 81], the "InternetAddress" attribute for a host would have a 32-bit value; Xerox Network Systems, on the other hand, use 48-bit internet host addresses [Dalal and Printis 81]. In some cases, an object may have several attributes of type "InternetAddress"; for instance, mapping host names to several addresses is useful for packet radio, multi-homed hosts, and partitioned networks [Cerf 79] [Sunshine and Postel 80] [Sunshine 82]. Additionally, for internetworks that support several diverse families of communication protocols, an attribute "SpeaksProtocols", whose value is a list of protocol types understandable by the named object, may be needed.

As an example of attributes at a higher layer, consider electronic mail systems that wish to name mail originators and recipients independent of the locations of their mailboxes [Garcia-Luna and Kuo 81] [Kerr 81] [Schicker 82] [IFIP 83] [Sirbu and Sutherland 84] These systems might use the name service to bind a user name to the name of the host computer on which his mailbox resides. In particular, the value associated with a "MailboxResidesAt" attribute would be a host name, which could then be presented to the name service to obtain the host's "InternetAddress" attribute. By modifying the value of their "MailboxResidesAt" attributes, users can change where they receive their mail without having to inform their correspondents.

1.2.4 Operations

The basic operation of a name service, then, is to map an object's name to attributes for that object. A simple operation to do this, *Lookup*, takes the name of an object and the desired attribute type and returns any attributes of the given type that are associated with the named object. Also, mechanisms must be provided to dynamically update the set of attributes for an object. For example, an *Update* operation might take a name and attribute as parameters along with an indication of whether the attribute should be added, removed, or modified in the name service database.

Additionally, name services may have special routines for manipulating group names, such as adding or deleting members; enumerating the individual members of a group can be an expensive operation if relegated to application programs, especially if groups contain other groups as members. The name service might wish to have operations that distinguish between aliases and preferred object names. Also, in order to guard against different objects being inadvertently assigned the same name, the object name should be registered with the name service independent of the object's attributes. In general, various operations on different types of objects and attributes may exist to facilitate type checking, access controls, consistency, and concurrency. The set of operations allowed by a name service can be as rich or baroque as those of any data storage facility. Furthermore, closely related services, such as authentication facilities [Needham and Schroeder 78], may be included in name services [Birrell *et al.* 82]. The clearinghouse client interface, providing dozens of operations [Oppen and Dalal 83], is a good example of the range of operations that may be desired.

1.3 The Thesis

Name services to support large distributed environments must themselves be structured as distributed systems. The advantages of distribution are well known: *modular growth* so that the name service can meet the needs of a continually expanding community, *availability* through using multiple processors so that the critical name services remain available to clients, *reliability* through redundancy so that valuable name service information is not corrupted, *autonomy* so that various organizations may cooperate in the high-level management of objects without compromising their internal security, and *performance* enhancements achieved through placing the name service information geographically close to where the interest in that data lies.

This dissertation develops a framework for building distributed name services to aid the management of objects in environments characterized as being large and diverse. The projected computing environment contains large numbers of networks of various technologies interconnecting a sizeable computing community. Vast numbers of diverse objects may be named and shared by members of the widely-distributed community; these objects come under the administrative control of a diversity of organizations participating in the environment. The facilities for storing and manipulating objects range from large mainframe computers to small personal workstations. Generally, end-to-end communication costs dominate the cost of interactions between distant sites. Environments of this sort are emerging with technological advances in computing and communications. The size and diversity of such computing communities place strenuous demands on name services.

The major thesis advanced and addressed by the research described herein can be simplistically stated as follows:

> *Physically distributed, but logically centralized, name services can be provided in a general and cost effective way, even for very large, geographically dispersed computing communities.*

A name service that supports this claim must solve the following principal problems:

- *Name resolution*: an object and its attributes may be stored at various, possibly several, locations in the internet; the name service must be able to determine these locations when presented with the object's name;
- *Administrative control*: administrative entities should govern the placement and protection of their objects; autonomous organizations cooperatively participating in the distributed community wish to retain control over the selection of trustworthy locations to store the attributes and names of their objects; particularly sensitive information should only be accessible by certain name service clients;
- *Overhead costs*: neither the size of the components of the name service at individual sites nor the number of interactions between components should be directly proportional to the size of the environment; although a name service may manage large numbers of objects and their attributes, small workstations with limited resources must be able to participate in and make use of the service;
- *Adaptation*: internet computing environments are continually evolving and expanding in size, either by the participating organizations acquiring new computing equipment, or by their interconnecting to other computing environments; the management of the name space must be flexible enough to gracefully adapt to changing demands;

- *Performance*: reasonable response times for accessing name services must be achieved; difficulties in obtaining reasonable response times arise due to the physical distribution of the environment and the cost of communication between distant sites; good performance is extremely important for the name service since it plays such a vital role in the overall system.

The next chapter describes existing name services and reflects on how they fail to solve some or all of these problems for large and diverse environments.

The remainder of this dissertation embarks on a path to substantiate the major thesis. Chapter 3 develops a basic architecture for distributed name services, providing a common framework in which later chapters address the principal problems. It broaches an important distinction between attribute data, information about named objects whose placement is controlled by administrative organizations, and configuration data, information managed entirely by the name service to locate attribute data. Chapter 4's examination of clustering to reduce the information needed in each name server for resolving names produces a general and powerful model of name resolution: structure-free name resolution. Prototype implementations of mechanisms for supporting this model are presented. Chapter 5 proposes a performance model of distributed name services that identifies factors contributing to the cost of name service operations. The model is applied to a sample environment to derive quantitative projections of the effect of name server placements, replicated data, and various assignments of authority on name service response times. Chapter 6 reports actual measurements obtained from the Grapevine registration service and uses them as input parameters to the performance model. Chapter 7 explores techniques for caching name service data at client sites to further enhance their performance. Treating caches as hints alleviates the cache consistency problem, while maintaining minimum accuracy levels guarantees performance benefits. Lastly, Chapter 8 recapitulates the principal problems outlined above along with their solutions.

Chapter 2

Name Service Designs: A Survey

Distributed name services have recently emerged in which a set of name servers collectively manage a global name space. The distribution of responsibility for parts of the name space, as well as the mechanisms for locating names, depends heavily on the name structure.

2.1 Existing Name Services

The desire to refer to objects by name and exchange information about these objects h as resulted in the development of network name services for several existing distributed environments. The major identifiable name services that have been implemented and documented are briefly summarized in the following subsections. Later sections of this chapter present in more detail the various aspects of these systems along with other proposals and designs for naming mechanisms.

2.1.1 NIC Name Server

The ARPANET [Roberts and Wessler 70], one of the first geographically-dispersed computer networks, has experienced a slow progression of name services. In the early days, the ARPANET Network Information Center (NIC) was established to maintain information about the network, including the master database of host names and their respective addresses. Every host stored a complete copy of this database, and the administrator of each host was responsible for updating its local copy when the master changed. This host table allowed members of the ARPANET community to name hosts, rather than refer to them by address, when transferring files between hosts or logging into a remote host.

With the growth in size of the ARPANET and its expansion into the DARPA Internet [Hinden *et al.* 83] [Cerf and Cain 83], maintaining up-to-date host name to network address mappings became increasingly difficult on individual hosts. The development of an experimental NIC Name Server slightly alleviated the situation by allowing the host table information to be retrieved incrementally via network protocols [Pickens *et al.* 79b]. This service eventually became the NIC Internet Hostnames Server [Harrenstien *et al.* 82]. The ARPANET host table stored by the server has been extended to include addresses for networks and gateways, as well as additional host information such as what protocols a particular host speaks, an indication of the services available, and what operating system it runs [Feinler *et al.* 82].

The NIC also provides services for obtaining personal information about ARPANET users. The NICNAME/WHOIS server supplies such information, including anything from a person's office phone number to his address for receiving electronic mail [Harrenstien and White 82].

2.1.2 DARPA Domain Name System

To this day, although the inadequacies of central administration are widely recognized by the DARPA Internet community, the master host table is still centrally maintained by the Network Information Center. Fortunately, plans are underway to switch over in the near future to a decentralized scheme for managing the host information [Postel 84]. The new Domain Name system will permit information on network entities to be distributed and replicated; the responsibility for its management will reside with the various administrative organizations comprising the DARPA Internet [Postel 84] [Mockapetris 83a] [Mockapetris 83b].

Included in the transition to decentralized name management is the adoption of the Domain Naming Convention [Mills 81] [Su and Postel 82] for naming electronic mail recipients as well as hosts. The Domain Naming Convention calls for a tree-structured name space in which each node of the tree has a label. The domain name of an object is simply the concatenation of the labels starting at the root and following a path through the tree; labels are listed from left to right and separated by dots. The Domain Name System stores information associated with each node of the tree as a set of "resource records" containing type, class, and data fields. It manages mailboxes, aliases, and group names in addition to the information currently maintained in the DARPA Internet host table.

2.1.3 BIND Server

The Berkeley Internet Name Domain (BIND) Server [Terry *et al.* 84] is an implementation of the DARPA Domain Name System for Berkeley UNIX. As such, it adheres to the Domain Naming Convention for identifying objects and to the basic set of operations designed for retrieving object attributes. However, unlike the Domain Name System which maintains a read-only database, it allows updates to the name service database to be applied dynamically using a primary update scheme with secondary snapshots for replicated data.

2.1.4 PUP Name Lookup Server

A decentralized name-lookup service was provided early in the development of the Xerox Pup Internet [Boggs *et al.* 80]. Servers on each network manage an identical database. Updates performed at any server are advertised to all other name servers using broadcast [Boggs 83]. This service still fills the needs of the PUP Internet, while Xerox' Network Systems [Dalal 82] have moved to a more decentralized clearinghouse service.

2.1.5 Grapevine

The Grapevine system [Birrell *et al.* 82] developed at the Xerox Palo Alto Research Center can be viewed as two systems in one, a mail system and a registration service. The latter provides name services designed primarily to support the mail system, including resource location, authentication, and access control. Names in the Grapevine environment identify mail recipients and are of the form, "F.R", where "R" is a registry name and "F" is unique within registry "R". Registries are intended to reflect organizational divisions.

A registration database that maps names to information about the names, including distribution lists and access control lists, is distributed and replicated among the many Grapevine computers. At this point in time, the Grapevine registration service might be considered the only regularly-used distributed name service.

2.1.6 Clearinghouse

The clearinghouse [Oppen and Dalal 83] is a decentralized service for locating named objects in a distributed environment. Like Grapevine, it was developed by Xerox, and the two systems have many things in common. In fact, clearinghouse's design was modeied after the Grapevine system except that clearinghouse names have three parts, "L:D:O" where "L" represents the local name, "D" the domain, and "O" the organization. The clearinghouse designers stress that domains and organizations, like registries in Grapevine, are logical rather than physical divisions.

Xerox's clearinghouse strives to serve as a general purpose binding agent. It maps an object's equivalence class, consisting of a distinguished name and associated aliases, into an arbitrary set of properties, where each property is an ordered tuple (PropertyName, PropertyType, PropertyValue). Clearinghouse's "properties" correspond to "attributes" as defined in Chapter 1. Property names, corresponding to attribute types, are standardized so that similar services can be easily identified. The only property types distinguished are "individual", an uninterpreted block of data, and "group", a set of names. The client interface supports many distinct operations for manipulating entities such as names, aliases, individuals, groups, and group members. Different operations on different types exist to facilitate type checking, access controls, consistency, and concurrency.

2.1.7 CSNET Name Server

One component of an effort to connect computer science research institutions with a long-haul computer network called CSNET was the development of the CSNET Name Server [Landweber *et al.* 83] [Solomon *et al.* 82]. Its primary function is to support mail applications, that is, aid in locating mail recipients. The CSNET Name Server maintains a centralized database containing keywords supplied by users to describe themselves. A mail recipient can be unambiguously identified in a location-independent way by supplying a suitable set of keywords, which are mapped by the server to a mailbox address "user@site". However, most mail users bypass the name service and simply use mail addresses directly. The major utility of the name service is in discovering the proper mail address of a particular person given descriptive information about him.

2.1.8 Cambridge Name Server

The Cambridge Distributed Computing System [Needham and Herbert 82] relies on a name server for translating unstructured names of services and machines into ring addresses. Roger Needham and Andrew Herbert describe the name server as "the most fundamental of all of the services provided by the distributed system" [Needham and Herbert 82]. In their environment, for instance, the name service operation is crucial for booting other services and for allowing a machine to discover its own address. When responding to service lookups, the name server indicates the protocol associated with the service, as well as the machine on which the service runs; however, the name server does not guarantee that the service is currently available. To achieve high reliability, the name server program, along with an initial name table, is stored in the read-only memory of a dedicated machine.

2.1.9 COSIE Name Server

The COSIE Name Server, designed and developed for use in a distributed office system [Terry 82], maintains a database of named attributes for an object. In order to support many different clients, the name server provides a very simple set of operations and places no restrictions on the syntax or semantics of the names it stores. It manages group names as well as individual names; group names have been used for lists of teleconferencing participants, mail distribution lists, generic services, and even to keep track of the users of a shared object (an alternative to reference counts).

2.1.10 R* Catalog Manager

The catalog manager for R*, a distributed database management system developed at the IBM San Jose Research Lab, maintains information used in distributed query processing. In addition to mapping names to the locations of database objects, it provides information about the objects such as the available access paths, their data schemas, the authorized users, and usage statistics. An object's system wide name has four components, "user@user-site.object-name@object-site". The "user@user-site" component permits different users to select object names that do not conflict, while the "object-site" component partitions the authority over objects. Name completion rules allow parts of the name to be left unspecified by database applications.

2.2 Structural Components

A general model has evolved for building name services in which a set of active entities called *name servers* share the responsibility for providing the service, while clients access the service through *name agents*.

2.2.1 Servers

Each *name server* manages part of the name space and runs on a single computer; interactions with other servers and clients transpire via the communication network. In the case of a centralized service, a single name server manages the complete name service database. Although several existing name services are provided in a centralized fashion [Harrenstien and White 82] [Harrenstien *et al.* 82] [Solomon *et al.* 82] [Terry 82], there is little argument that name services to support large and diverse computing environments should themselves be organized as distributed systems [Clark 82].

In a distributed name service, several name servers collectively manage the name space and support the basic set of operations. Generally, the name servers act as peers in that they all play an identical role in the system. That is, the function of the service is not partitioned among servers; the control and data are simply decentralized. All name servers present a common interface and accept operation requests from any client, though the contacted name server may not contain enough information to process the operation locally. Grapevine, the clearinghouse, and the Domain Name System are all organized in this manner.

Differing attitudes exist as to whether the name service should use dedicated machines or run on hosts along with other services and clients. For instance, the CSNET Name Server is a dedicated host, and the Grapevine system runs on a collection of dedicated machines. On the other hand, the R* distributed data management system, including its catalog management component, executes on all hosts. The V-System, developed at Stanford University, adopted a policy where each server for a class of objects provides the name service for those objects; thus several object-specific name servers might reside on a workstation [Cheriton and Mann 84]. The Cronus Distributed Operating System also requires a name server on every machine, but for availability reasons; the designers argue that "it should be possible to access an object when the site that stores the object is accessible" [Hoffman *et al.* 83].

2.2.2 Agents

Clients of the name service prefer to be unaware of its distributed nature, and hence interact with *name agents* that assume responsibility for communicating with remote name servers. Name agents thus act as intermediaries between name servers and their clients, allowing client programs to be written as if the name service were locally available.

The notion of a name agent has been provided in several systems under various names.

The Grapevine system has similar components called "GrapevineUser" [Birrell 83], the COSIE Name Server calls them "user interfaces" [Terry 82], the DARPA Internet Domain Name system has "resolvers" [Mockapetris 83a], the CSNET Name Server uses "name server agent programs" [Solomon *et al.* 82], one proposal calls for "application interface processes" [Su 82], and the clearinghouse requires "stub clearinghouses" to be resident in every client [Oppen and Dalal 83].

In cases where a name server and its clients reside on the same machine, as would arise with policies that require a server on every host, the clients' name agents might be unnecessary. However, besides speaking the proper communication protocols, name agents may perform additional functions such as maintaining a detailed knowledge of the name space and of existing name servers. One proposal suggests using name agents to negotiate for resource availability and compatibility once a resource manager is located through the name service [Su 82]. Chapter 7 addresses the issues of caching the results of recent name service queries within name agents.

The interface provided by a name agent to its clients may mimic the interface provided by the name servers, or may be tailored to a particular application. "Value-added" services provided by the name agent, such as caching or resource negotiation, undoubtedly require interfaces to new operations.

Each name service client most likely utilizes a single name agent. However, each name agent may either serve a single client or be shared by different clients in the same locale. These two organizational choices are depicted in Figures 2.1 and 2.2.

If the name agent is structured as a set of subroutines that are simply linked into the client program, then each client has a private name agent. On the other hand, a name agent that is shared among clients may be incorporated into the operating system kernel, with system calls used to invoke name service operations, or may exist as a separate process and be accessed via an interprocess communication (IPC) mechanism. For example, the initial BIND name agent, a domain name resolver, was implemented as a set of C language library routines [Terry *et al.* 84]; current efforts are underway to migrate the resolver to a separate UNIX process so that a shared cache can be maintained by the name agent.

2.3 Functional Components

A name service can be functionally decomposed into three components: communication, database management, and name management. A name service must be able to store data reliably and communicate among servers and between servers and agents. Name management builds upon database and communication technology to allow the distributed name service database to be queried and modified.

2.3.1 Communication

Name servers and name agents reside on various machines distributed throughout the environment and hence must rely on a communication protocol for their interaction. The usual three styles of communication exist for the server/agent and server/server protocol: using self-contained *datagrams* for exchanging data, establishing *virtual circuits* to transmit byte-streams, or employing *remote procedure calls* to invoke remote operations in a similar manner to local ones. Selecting the proper protocol involves weighing the cost of the protocols against the benefits they provide. For instance, datagrams are generally unreliable, though less expensive than virtual circuits, which provide reliable data transmission. Remote procedure calls are conceptually simple to use because of their resemblance to local procedure calls; nevertheless, the request-response paradigm enforced by remote procedure calls may not always be desirable.

In practice, different protocols may be desired for different modes of communication taking place between name servers and agents. For example, reliable communication may be unnecessary for

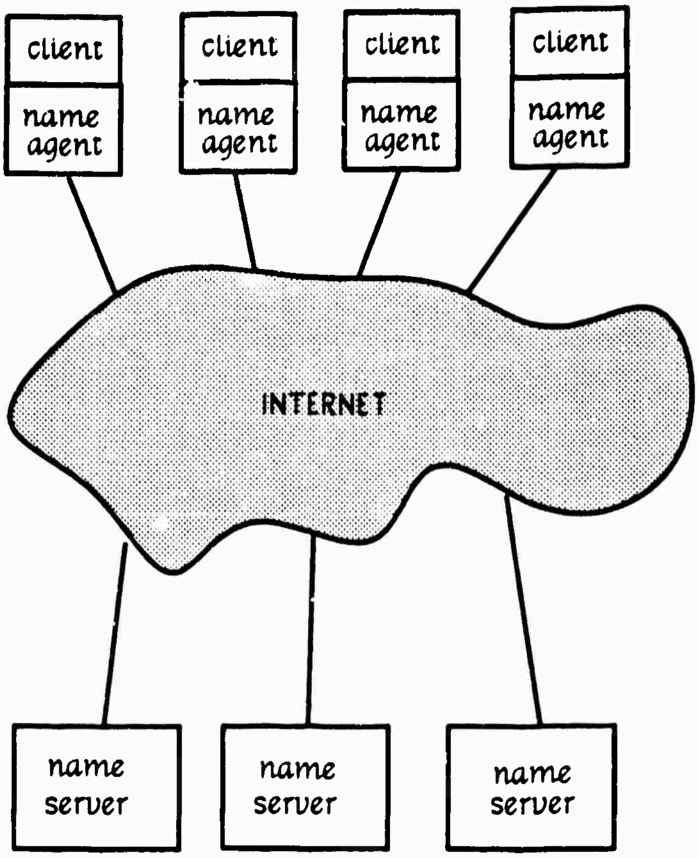


Figure 2.1: **Individual name agents.**

Lines represent loosely coupled interfaces while common edges represent tightly coupled interfaces.

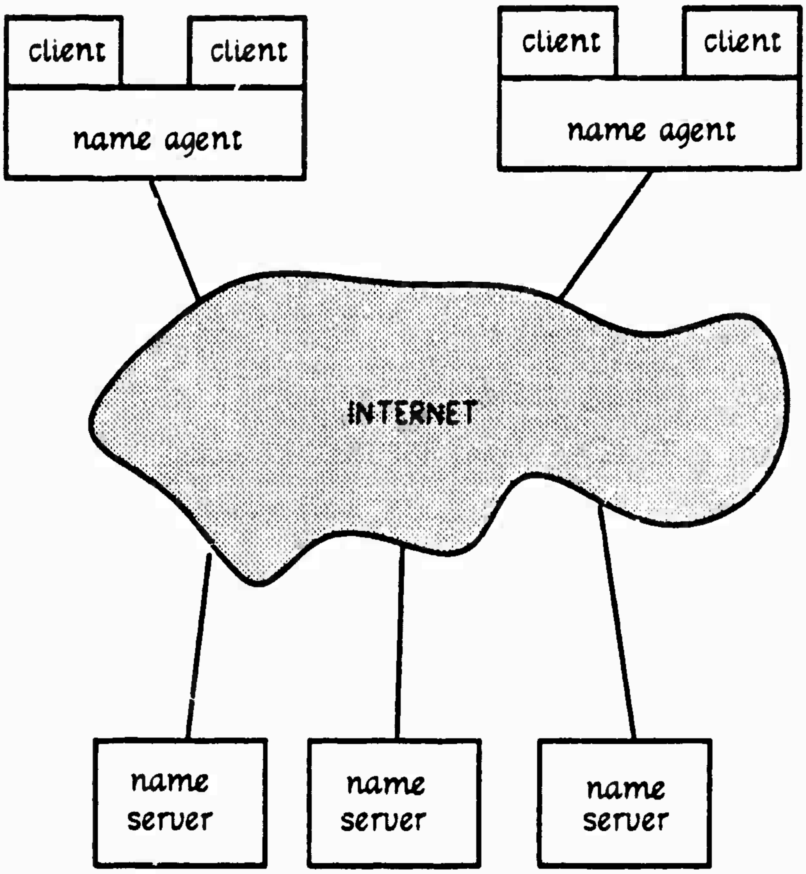


Figure 2.2: Shared name agents.

Lines represent loosely coupled interfaces while common edges represent tightly coupled interfaces.

invoking name service operations since they can be easily made *idempotent*, allowing the operations to simply be retried in case of communication failure. However, critical communication, such as the exchange of authoritative data between servers, should be reliable.

Grapevine [Birrell 83] and R* [Lindsay *et al.* 84] use byte stream protocols for communication so that the cost of authenticating communicants can be completely incurred at connection establishment. The DARPA Internet, on the other hand, has traditionally used datagrams for invoking name service operations [Harrenstien 77] [Postel 79]. The Domain Name System, however, specifies that a virtual circuit should be established if the name service response is too large to fit in a single datagram [Mockapetris 83b]. It also uses virtual circuits for reliably propagating updates to replicated data. To accommodate a diversity of clients, the CSNET Name Server accepts queries in a variety of forms, including electronic messages [Solomon *et al.* 82].

2.3.2 Database management

One of the major responsibilities of a name service deals with managing the name service database of objects' attributes. A lot of work has been done by the database community in developing techniques for query processing, concurrency control, and transaction management [Gray 78]. However, surprisingly enough, the COSIE Name Server [Terry 82] is the only one of the services discussed in this chapter that uses a general purpose database management system to store its information (aside from data dictionaries); perhaps because database management systems have reputations for being big and slow, perhaps because complex query languages are not needed to support the simple name service operations, perhaps because name services have very simple data schemas.

Although database transactions [Lampson 81] are useful for implementing atomic name service operations, reliable data storage may not always be necessary. For example, the COSIE Name Server [Terry 82] makes a distinction between temporary and permanent objects. Updates to attributes of permanent objects use the underlying database management system, while temporary object information is placed in the in-core buffer pool, but never committed to the resident database. Registering temporary objects is thus faster than registering permanent objects since a database transaction is not required. Registering objects as temporary is useful for processes that rendezvous through the name server or for distributed programs that are being debugged. In both cases, the permanence of the information is neither required, nor desired. For example, programs that are being debugged often fail in ways that prevent them from unregistering themselves with the name service; if registered as temporary, the information associated with these programs is automatically purged from the name service database when its buffer storage is reused.

Several techniques for managing replicated data in a distributed computing environment have been proposed and thoroughly discussed in the literature. Bruce Lindsay *et al.* [Lindsay *et al.* 79] and Elmar Holler [Holler 81] provide good overviews of these techniques. These general algorithms for maintaining consistent copies of replicated data can be adopted for the distributed management of name service information. However, they assume no knowledge about the semantics of the data being managed. Researchers at Carnegie Mellon University developed a special algorithm for replicated directories based on Gifford's weighted voting [Gifford 79] that takes advantage of the properties of name directories to achieve high availability and performance [Daniels and Spector 83] [Bloch *et al.* 84]. Basically, they achieve higher concurrency by dynamically partitioning the set of names stored in a directory and maintaining a version number for each partition.

Also, general replicated data algorithms, such as weighted voting, almost exclusively consider strong consistency to be important. The designers of the Grapevine system argue that name service clients can cope with temporary inconsistencies. Much of the work in the design of the Grapevine registration service was in designing an algorithm for replicated data that exploits the semantics of registration data [Birrell *et al.* 82] [Schroeder *et al.* 84]. The Grapevine system has a weak notion of consistency among the various replicated copies of a registry. Availability is enhanced by allowing updates to a registry to be performed at any site and then propagated to all other storage sites. The only guarantee is that all of the copies will eventually converge to a consistent state. Active and deleted sublists of entries, as well as timestamps, must be maintained in order to merge copies that have been simultaneously updated. However, conflicting simultaneous updates are not guaranteed to be resolvable. Greg Thiel developed similar algorithms for merging replicated database catalogs that have been independently updated during a network partition [Thiel 83]. Again, the goal was to improve update availability by reducing the consistency requirements.

Lastly, many algorithms for replicated data assume that all data storage sites are always able to communicate with each other. However, for dialup networks with very loose topologies, such as UUCP [Nowitz 78] or CSNET's PhoneNet [Comer 83], servers may only be able to exchange updates at limited times. For this reason, the BIND Server uses a primary update scheme in which the responsibility for requesting updates lies with the secondary servers [Terry *et al.* 84]. For simplicity, all updates are directed to a primary server, which transfers incremental updates to secondary servers upon request. The restriction that updates get directed to a single server eliminates the need for merge algorithms, but reduces update availability and concurrency.

2.3.3 Name management

Several schemes for naming objects have been proposed, though few of the proposals have addressed the issues of distributed name management. The major aspects of name management include *name distribution*, the assignment of authority for parts of the name space to various name servers, and *name resolution*, the mechanism for locating the attributes of a specific object given its name. Generally, the structure of names influences the way in which they are resolved and distributed.

Many naming mechanisms trivialize name management by utilizing centralized name services. Others, such as the Pup name service [Boggs 83] or the Mininet system [Livesey 79], fully replicate the name service information in all servers; name resolution is thus unnecessary since any name server is able to respond to any lookup request.

Some proposals allow the name service database to be partitioned and distributed, but rely on broadcast or searches of name servers to find information. Such a protocol for locating resources in the DARPA Internet has been recently proposed [Accetta 83]. Often, the name service database is distributed such that each name server manages local objects. References to local objects can then be resolved by consulting the local name server; resolution of names for nonlocal objects resorts to using broadcast [Janson *et al.* 83] [Lyngbaek and McLeod 82] [Gelernter 84]. Bremer and Drobnik carry this a step further and suggest a scheme in which the environment is divided into regions where regional directories maintain name-to-address mappings for all objects residing in their region [Bremer and Drobnik 79]. Name resolution proceeds in three steps: the local name server, which may contain incomplete information, is consulted; if the desired name is not found, then a regional server is contacted; if that is unsuccessful, then a request is broadcast to all other regions.

To avoid broadcast but permit distributed data, many systems incorporate an object's network location into its name and adopt the policy that a local name server manages local objects [Lyngbaek and McLeod 82] [Chou *et al.* 83] [Cheng 84] [Curtis and Wittie 84b]. These *location-dependent names*, of the form "local-name@machine", carry with them the information necessary for name resolution. Mail systems, including those used in the DARPA Internet and CSNET, have traditionally accepted such names for identifying mail recipients. RSEXEC, perhaps the first attempt to create a network-wide name space for objects other than mailboxes, used this approach to refer to files on TENEX machines scattered around the ARPANET [Thomas 73].

The R* system requires each catalog manager to maintain information about all locally stored objects and all objects that were created locally [Lindsay 80]. Names are of the form, "object-name@object-site", where the "object-site" represents the birthsite of the object, not its storage site. These might be called *authority-dependent names* since an object is allowed to migrate to other sites, but its birthsite remains the authority for the object. The birthsite must track the object's movements so that its name can be resolved. Debra Deutsch also proposed using birthplaces as a means for distributing and locating information about mail recipients [Deutsch 79].

The V-system also uses authority-dependent names, but manages them in a slightly different manner: each server for a class of objects manages the names for those objects [Cheriton and Mann 84]. In order to allow a uniform way for interpreting object names, all names are prefixed by the server identifier. Names of the form "server.object-name" are resolved by first contacting a local "context prefix server" that indicates where to forward the resolution request; different servers can resolve the "object-name" in different ways, though many use hierarchical name spaces with nested contexts.

Systems, such as Grapevine or the Domain Naming System, use *location-independent names*, sometimes called *domain names* or *organizationally-partitioned names*. In these systems, an object's name is only indirectly associated with the server or servers that manage information about the object.

In the Grapevine system [Birrell *et al.* 82], registries represent the granularity for partitioning and replication of the registration data; that is, a registry is treated as an indivisible unit when it comes to storage site selection. Registries can be replicated in several servers, and a given server may manage more than one registry. A special registry, which is replicated in every registration server, enables any Grapevine server to determine which servers contain the database entries for a particular registry. Since object names explicitly contain the registry in which the object resides, all name lookups require two steps: first the authorities for the name's registry are discovered, then one of them is contacted. Clearinghouse's distributed lookup algorithm is basically the same as Grapevine's except that name resolution takes place in three steps since clearinghouse names have three parts instead of two [Oppen and Dalal 83].

The Domain Naming System [Mockapetris 83a] [Mockapetris 83b] partitions the name space into "zones". A zone can be specified by the domain name of its root and the names of its endpoints. If an endpoint of a zone is not a leaf node, then that node serves as the root of another zone. Zones represent the administrative divisions within the name space. For example, Figure 2.3 indicates a couple of zones that might exist on the Berkeley campus. As with Grapevine registries, zones are indivisible units of storage, and a many-to-many mapping may exist between zones and name servers. Thus, the boundaries between zones indicate possible delegations of authority. The Domain Naming System resolves names a label at a time starting at the root and traversing down the branches of the tree. The resolution of a name migrates from server to server in accordance with the delegations of authority until all labels of the name have been examined. As an optimization, if a server receives a name lookup request for a name that is in one of its zones or a zone that it has delegated authority to, the resolution of the name need not start at the root of the tree, but rather can start at the root of a zone in which the domain name of the root is a prefix of the name being resolved.

The Cronus [Hoffman *et al.* 83] and LOCUS [Popek *et al.* 81] [Walker *et al.* 83] distributed operating systems also support tree-structured symbolic object names. LOCUS has the notion of "file groups" that correspond to zones; it maintains a network-wide "mount" table for resolving names. The Cronus designers adopted a policy in which a "dispersal cut" is made through the name space such that the "root portion" is fully replicated at all sites, and entire subtrees below the cut are stored within a single site. In other words, the entire name space above the cut is a single zone, and subtrees below the cut represent individual zones, as depicted in Figure 2.4. This enables names to be resolved by contacting at most two name servers.

2.4 Performance Issues

The existing work on name services stresses functionality, while performance considerations have remained of secondary importance in most work to date.

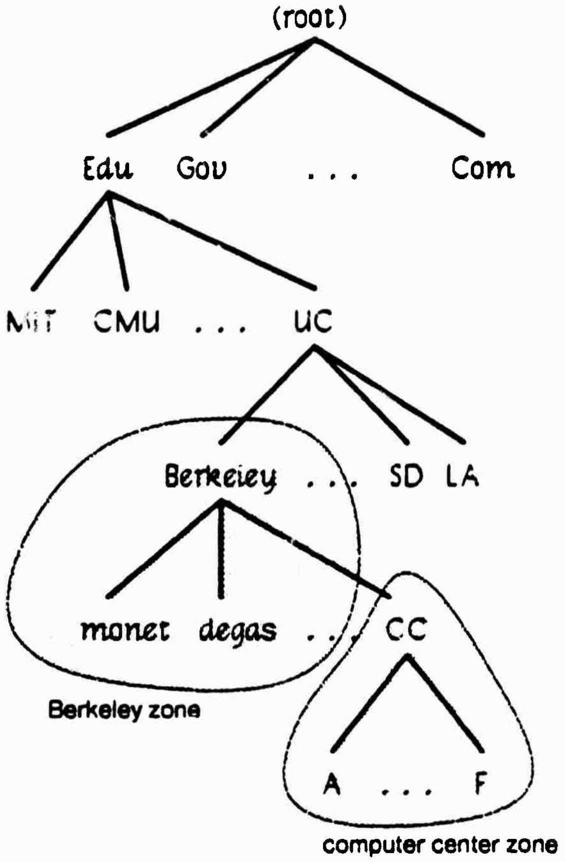


Figure 2.3: Domain name space with sample zones.

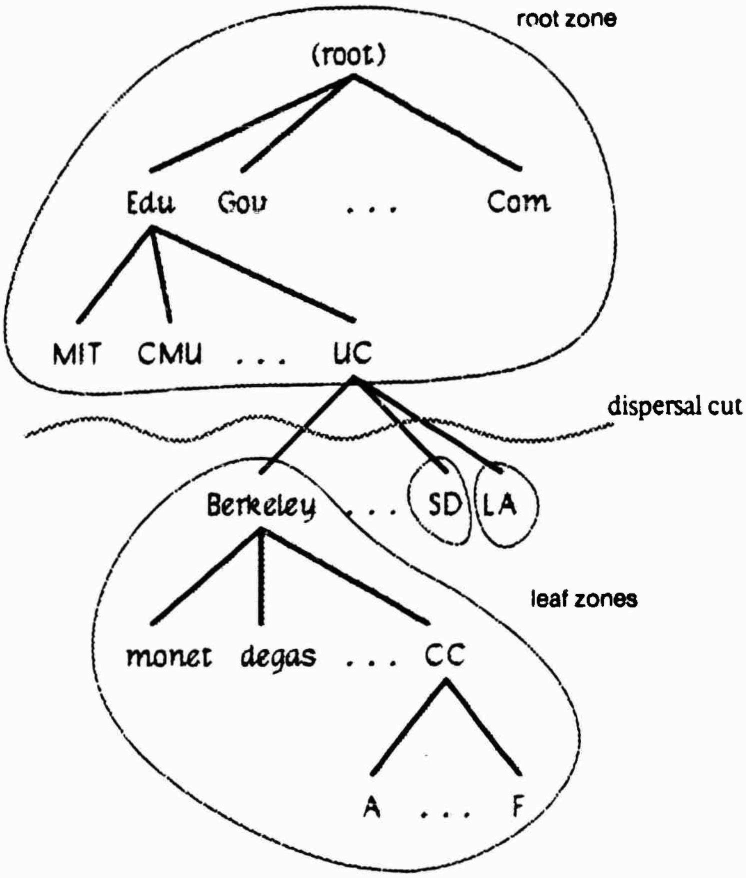


Figure 2.4: **Hierarchical name space with dispersal cut.**

2.4.1 Models

Performance models of name services have been noticeably lacking. Typically, the name service designers or administrators distribute the name space among servers according to their intuition or experienced observations of the environment rather than modeling various alternatives. The few recent attempts to analyze distributed name management schemes have been concerned with very simple strategies.

Yen-Yi Wu studied file directory systems for locating files in networks with either loop or star configurations [Wu 83]. The directory schemes conside ed include centralized directory data, fully replicated directory data, and some hybrid schemes bas d on localized authority and searches. Wu's model allo ed expected query response times for the various directory schemes and network configurations to be computed.

The only known paper that discusses the performance of name services in an internet environment proposes having regional name servers manage a two-part name space in a hierarchical fashion [Chou *et al.* 83]. All regional servers store complete information about objects in their local network; updates are propagated by broadcasts. Chou *et al.* introduced a network communication model, which was used in simulations to analyze the cost of this proposed distributed update scheme for high transmission error rates.

2.4.2 Measurements

Measurements of distributed computer systems invariably provide needed insights into their operation and suggest ways of improving their performance. Of the name services discussed in this chapter, only the Grapevine system manages a partitioned and replicated name space with a large user community. Other emerging name services, such as the DARPA Domain Name system, should benefit from experiences with Grapevine. As the Grapevine designers put it, "There is no alternative to a substantial user community when investigating how the design performs under heavy load and incremental expansion" [Birrell *et al.* 82]. Some measurements and experiences with Grapevine have been recounted concerning the administration and reliability of the system [Birrell *et al.* 82] [Schroeder *et al.* 84]; no work has been identified in which measurements were obtained to aid in configuring name services.

2.4.3 Caching

A couple of present-day name services, Grapevine and the R* catalog manager, employ caches to improve the performance of name service lookups. Grapevine message servers cache hints about individuals' preferred inbox sites; out-of-date cache entries are easily detected when servers attempt to deliver a message to a moved or deleted mailbox [Birrell *et al.* 82]. R* database sites use locally cached catalog entries in distributed query planning; when the formulated plan is distributed to the sites involved, version numbers for the catalog entries on which the plan is based can be compared against the current catalog entries to determine the validity of the plan [Lindsay 80]. Other systems have suggested the use of caches, but concrete designs have yet to emerge.

2.5 Evaluation of Previous Work

Significant work has been done in the area of communication protocols for accessing name services and in the area of database management systems for storing object attributes. The currently unresolved problems in designing name services concern how to manage large distributed name spaces.

Contemporary name services are emerging in which the attribute information is both distributed and partitioned. These planned or existing systems make substantial contributions to the general techniques needed to build distributed name services. Nevertheless, all of the existing designs fail to adequately address some of the problems outlined in Section 1.3 for very large and diverse computing environments:

- *Name resolution*: All name services are able to resolve unambiguous object names in one way or another. In existing name services that do not rely on broadcast, the process of resolving names is driven by a name's syntactic structure and dependent on how names are distributed among name servers. Name resolution always proceeds by successively resolving individual labels of a name. Unfortunately, existing name services' reliance on syntactic structure in order to locate an object or its attributes place constraints on the management of the name space; these constraints prevent solutions for some of the other principal problems from being realized.

  As an extreme example, location-dependent names restrict the mobility of an object once a name has been assigned. Changing the name of an object is an expensive operation since all of the references to the named object become invalid; hence, object names are generally considered permanent. Location-dependent names force objects to change their names in order to relocate.

- *Administrative control*: Even authority-dependent and existing location-independent naming schemes provide less than perfect administrative control over the placement of an object's attributes. All current name services distribute the authority for names to various servers based on the structure and contents of the name; syntactically similar names, for some similarity criteria, have the same authorities. For example, in the Grapevine system, all of the names belonging to a particular registry have the same set of authoritative name servers; in the Domain Naming System, a name's zone determines its authorities. Because of the syntactic distribution of names in existing systems, the assignment of a name to a new object is partially governed by an organization's concerns for the name servers that store the object's attributes. Changing an object's name servers requires changing its name or assigning new name servers for all objects in the same syntactic partition of the name space.

- *Overhead costs*: Name management schemes in which the entire database is maintained by a single name server place an unreasonable load on the server, when it is used in large environments, due to the storage requirements and the frequency of updates. A few existing name services are able to successfully manage large numbers of objects by partitioning the name space among many servers. A potential difficulty arises, however, for naming conventions with a fixed number of levels. Grapevine, for example, with its two-part name space, requires all servers to know about all registries; truly enormous computing communities would require a significant number of registries. The clearinghouse and R* catalog manager face similar problems.

  A lack of scalability also represents a major failing of systems that rely on broadcasting name resolution requests to all name servers. Although David Boggs claims that any network should provide broadcast mechanisms [Boggs 83], the cost of such mechanisms for large internetwork environments renders full broadcasts infeasible.

- *Adaptation*: The inability to adapt to growing communities with changing requirements is the main deficiency of traditional name management techniques. Existing name services, whose basic mechanisms have such a strong reliance on the syntactic structure of the name space, may lack the flexibility to scale up to very large environments. At best, the system administrators that configure the name service initially must carefully partition the name space according to the projected growth of the environment so that no partition becomes unmanageably large. Name services should be able to be reconfigured if the present servers become overworked or overburdened with data. With current services, reconfiguration occasionally requires objects to change their names because the name space is distributed among servers according to syntactic partitions. As an example of a lack of flexibility, as a Grapevine registry grows over time, no provisions can be made for dividing its data between different name servers. At least one Grapevine registry has already been split, causing some of its members to be renamed.

- *Performance*: As indicated earlier, very few studies have attempted to measure or predict the performance of name service operations. Within the framework of most name services, decisions must be made concerning how to distribute and replicate parts of the name space; these decisions

> drastically affect the response times for name service lookups or updates. The Grapevine designers have provided some suggestions based on their experiences, but measurement and modeling tools are really needed to aid in configuring large name services. The utility of techniques such as caching and data replication can only be determined once the operation of a name service is fully understood, including clients' referencing behavior.

The DARPA Internet's Domain Naming System seems to come the closest to handling very large and diverse computing environments, though it has yet to become fully operational. This dissertation adopts many of the architectural properties of such a service, but develops a more flexible approach to name management: *structure-free name management* breaks the strong ties between the structure of names and their management.

Chapter 3

Name Distribution

A basic architecture for distributed name services provides the framework in which to explore the problems of managing large name spaces. Facilities for internetwork communication and for maintaining replicated and distributed copies of data serve as the foundation for building distributed name management mechanisms. Structure-free name distribution, achieved by introducing a special attribute that indicates each object's responsible name servers, permits more flexible assignments of authority than those based on the name structure.

3.1 Foundations

3.1.1 A Layered Architecture

This dissertation develops an architecture for building distributed name services, including mechanisms for distributing, resolving, and caching names. As in current name service designs, several name servers collectively manage the name space and support the basic set of operations. The facilities required of each name server can be organized in layers as depicted in Figure 3.1. Subsequent sections describe each of these layers in more detail as well as the interactions between layers.

Segments of programs to implement the name management mechanisms are provided in places in order to make the architecture concrete and present guidelines for future implementors of distributed name services. The programs are written to be easily understandable, not to be efficient or complete implementations. The casual reader concerned with simply understanding the concepts presented should be able to skip the program segments; though they often help to clarify the discussion.

All of the program examples are presented in the Mesa programming language [Mitchell *et al.* 79]. The intent is that the reader need not be familiar with Mesa in particular; familiarity with constructs common in block-structured languages should suffice for understanding the examples. Explanations of unconventional or esoteric language facilities are given in the footnotes.

3.1.2 Communication Support

The examples presented throughout this dissertation utilize a hypothetical remote procedure call mechanism that allows procedures to be executed reliably on remote machines. Its use requires adding a new NETADDRESS data type to the programming language, which is the internet address of the host on which the called procedure is to be executed, and a new primitive, AT , which binds the call to a particular address. For instance,

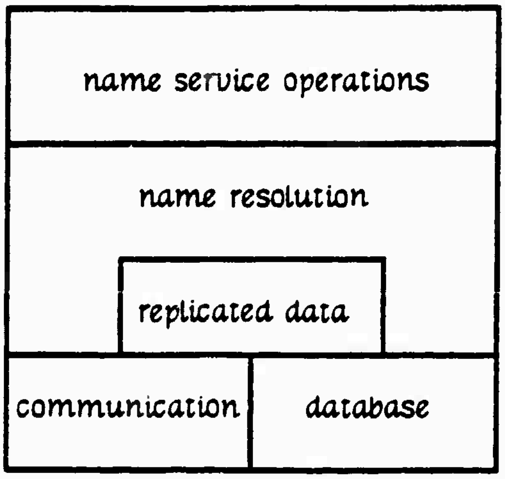


Figure 3.1: Functional layers in a name server.

```
address: NETADDRESS;
. . .
result ← Module.Procedure[args] AT address;
```

invokes the given procedure of the given module at the specified host address and waits for the result to be returned. This assumes an internetwork environment with a global address space from which values of type NETADDRESS can be drawn.

The use of the AT operator is introduced to explicitly indicate the interactions between programs running on separate machines. Such a facility does not actually exist in the Mesa programming language. Nor would a real remote procedure call mechanism be incorporated into the language in this manner since remote procedure calls are generally intended to look identical to local procedure calls with the bindings between servers and clients being performed by the runtime package [Birrell and Nelson 84].

Remote procedure calls were selected so that the semantics of the communication can be presented in an easily understandable way without being concerned with the details of a particular communication protocol or package. Furthermore, the details of packing operations, their parameters, and their results into messages can be ignored.

3.1.3 Database Support

The *name service database*, containing attributes for the universe of named objects, is distributed and replicated among the name servers. A given attribute may be managed by one or more name servers. However, for simplicity, all of the attributes belonging to a given object should be maintained together. That is, if a name server stores one attribute for a named object, then it stores all attributes for that object. The name servers that store information about a particular object, and assume responsibility for reliably managing that information, are called the *naming authorities* or *authoritative name servers* for that object.

```
Database: DEFINITIONS IMPORTS NS = BEGIN

AttributeTuple: TYPE = RECORD[
    name: NS.Name,
    attribute: NS.AttributeType,
    value: NS.AttributeValue
    ];

DatabaseObject: TYPE = LIST OF AttributeTuple;

Query: PROCEDURE[db: DatabaseObject, name: NS.Name, attribute:
    NS.AttributeType] RETURNS[AttributeTuple];

AddTuple: PROCEDURE[db: DatabaseObject, tuple: AttributeTuple];

DeleteTuple: PROCEDURE[db: DatabaseObject, tuple: AttributeTuple];

ModifyTuple: PROCEDURE[db: DatabaseObject, tuple: AttributeTuple];

TupleID: TYPE ; -- opaque type

Enumerate: PROCEDURE[db: DatabaseObject, next: TupleID]
    RETURNS[tuple: AttributeTuple, next: TupleID];
END.
```

Figure 3.2: **Database interface.**

3.1.3.1 Local database management

Each name server uses a database management system to store a set of *attribute tuples*, each consisting of an object's fully qualified name along with an attribute type and value. Attribute tuples are maintained by the database management system in special *database objects*. Figure 3.2 presents the interface for the **Database** module that provides facilities for storing and retrieving attribute tuples[1].

The **Query** operation retrieves the attribute tuple with a given name and type from the specified database object. An attribute type of "ANY" may be given, indicating that any attribute for the named object may be returned. **AddTuple** inserts the given tuple into the database object, while **DeleteTuple** removes a tuple from the database object. **ModifyTuple** performs an atomic update to a database attribute tuple; that is, it looks for a tuple whose name and type matches the parameter tuple and replaces its value. Finally, **Enumerate** allows the contents of a database object to be retrieved a tuple at a time; a parameter indicating the next tuple to return may be given as NIL to start the enumeration.

Protection of database objects, that is, the right to change existing attributes or add new attributes to an object, is enforced by the underlying database management system. The database interface

[1] This module makes explicit use of type declarations from the name server interface, **NS**, presented later in Figure 3.4. The **list of** construct is actually an extension to Mesa present in the Cedar programming language.

```
Replicated: DEFINITIONS IMPORTS NS, Database = BEGIN
OPEN Database;

StorageSites: TYPE = LIST OF NETADDRESS;

Query: PROCEDURE[sites: StorageSites, db: DatabaseObject,
    name: NS.Name, attribute: NS.AttributeType]
    RETURNS[AttributeTuple];

AddTuple: PROCEDURE[sites: StorageSites, db: DatabaseObject,
    tuple: AttributeTuple];

DeleteTuple: PROCEDURE[sites: StorageSites, db: DatabaseObject,
    tuple: AttributeTuple];

ModifyTuple: PROCEDURE[sites: StorageSites, db: DatabaseObject,
    tuple: AttributeTuple];

Enumerate: PROCEDURE[sites: StorageSites, db: DatabaseObject,
    next: TupleID] RETURNS[tuple: AttributeTuple, next: TupleID];
END.
```

Figure 3.3: **Replicated data interface.**

presented is oversimplified in that it does not show the parameters needed for protection, error handling, and transaction management. Although these are important issues being tackled by the database research community, they are not discussed in this dissertation.

Generally, a database management system resides on the same machine as each name server. However, the database support could come from separate database machines accessed via the remote procedure call protocol, as long as they support the **Database** interface.

3.1.3.2 Replicated data

An object with several authoritative name servers has its attributes replicated among those servers. Name service operations thus require the participation of possibly several machines in order to read or update replicated database tuples. Complete up-to-date copies of the object's attributes could be stored by all authorities, necessitating a *Read-any/Write-all* algorithm for replicated data. Alternatively, a more elaborate scheme, such as weighted voting [Gifford 79], could be used to maintain consistent replicas.

Rather than attempting to choose a particular algorithm for maintaining consistency among replicated database objects, this dissertation presumes the existence of a **Replicated** module providing the interface given in Figure 3.3. The operations allowed on replicated database objects are identical to those provided by single-site database managers. The replicated operations merely take an additional parameter indicating the storage sites of all copies.

Using a *Read-any/Write-all* scheme. the replicated query routine would simply be

```
Query: PROCEDURE[sites: StorageSites, db: DatabaseObject,
name: NS.Name, attribute: NS.AttributeType]
RETURNS[AttributeTuple] = BEGIN
     address· NETADDRESS ← SelectSite[sites];
     Database.Query[db, name, attribute] AT address;
     END;
```

The choice of a particular server to direct the operation to, as embodied in **SelectSite**, should be based on some criteria such as cost, closeness, or availability. Choosing randomly from the list of storage sites has the nice property that no knowledge of other servers is required. Nevertheless, as demonstrated in Chapter 5, substantial performance benefits can be obtained if the server is selected intelligently. Thus, servers may wish to know what fellow servers are currently operational, how expensive cross communication is, and how busy other servers are. A name server could acquire such information by exchanging status information with other servers or by consulting local routing tables to determine how close servers are to one another.

As another example of an instantiation of the replicated data module, consider a weighted voting scheme. Using the **CollectReadQuorum**, **CollectWriteQuorum**, and **SelectFastestCurrentRepresentative** routines from Dave Gifford's prototype implementation [Gifford 79], the operations to retrieve and modify a database attribute tuple could be implemented as follows:

```
Quorum: TYPE = StorageSites;

Query: PROCEDURE[sites: StorageSites, db: DatabaseObject,
name: NS.Name, attribute: NS.AttributeType]
RETURNS[AttributeTuple] = BEGIN
     readq: Quorum ← CollectReadQuorum[sites];
     best: NETADDRESS ← SelectFastestCurrentRepresentative[readq];
     Database.Query[db, name, attribute] AT best;
     END;

ModifyTuple: PROCEDURE[sites: StorageSites, db: DatabaseObject,
tuple: AttributeTuple] = BEGIN
     writeq: Quorum ← CollectWriteQuorum[sites];
     WHILE writeq # NILDO
          Database.ModifyTuple[db, tuple] AT writeq.first;
          writeq ← writeq.rest:
          ENDLOOP;
     END;
```

Notice that the query routine is similar to that of the previous approach, except the selection of a site from which to retrieve the desired data is confined to those sites belonging to the read quorum with up-to-date copies; the database management system must maintain version numbers for the data so that current representatives can be determined.

3.2 Structure-free Name Distribution

3.2.1 Assigning authority

For large computing environments, not all name servers can be authoritative for all objects; the authority for objects must be divided among servers according to administrative concerns. The various organizations sharing a common name space desire flexibility in configuring the distributed name service, that is, choosing the authorities for an object. This dissertation proposes *structure-free name distribution*, which places no restrictions on the administrative control over parts of the name space. In particular, the owner of an object may choose its authoritative name servers, subject to administrative constraints, independent of the object's name.

This differs from existing name services, which distribute names to authoritative servers based on syntactic characteristics of the names, as described in Chapter 2. Syntactic distribution of the name space generally fails to satisfy the desires for strong administrative control and graceful growth. Recall that, with location-dependent and authority-dependent names, an object's authority is directly represented in its name so that changing the authority requires changing the object's name, a prohibitively expensive operation.

Systems that use location-independent names assign authority based on zones; what zone an object's name belongs to, based on syntactic characteristics of the name, determines the object's authorities. Structure-free name distribution can be considered a scheme in which each object belongs to its own zone. This permits maximum flexibility in the administrative assignment (and reassignment) of authority. It also simplifies name management since name servers need not agree on what zones make up the name space.

3.2.2 Authority Attributes

In order to resolve names in a distributed environment, the name service must be able to determine the authoritative name servers for every named object. This can be accomplished by maintaining *configuration data* that contains lists of the authoritative name servers for every object. Such data is stored in the name server database as attribute tuples of type "Authorities":

```
ServerName: TYPE = Name;

AuthorityList: TYPE = LIST OF ServerName;

-- AttributeType = "Authorities" --
AuthoritiesData: TYPE = AuthorityList;
```

Essentially, an object's naming authorities are attributes of that object, though these attributes are treated specially since they are used solely by the name service; authority attribute tuples are not stored with the rest of an object's attributes.

Conceptually, authority attributes comprise the configuration database used for name resolution,

```
configurationDB: Database.DatabaseObject;
```

Assuming all name servers store the complete set of configuration data, name resolution involves a single database query,

```
Resolve: PROCEDURE[name] RETURNS[AuthorityList] = BEGIN
    authorities: AuthoritiesData;
    tuple: Database.AttributeTuple;
    tuple ← Database.Query[configurationDB, name, "Authorities"];
    authorities ← LOOPHOLE[tuple.value, AuthoritiesData][2];
    END;
```

However, for very large and diverse environments, the configuration database is undoubtedly too cumbersome to be stored everywhere in its entirety. The next chapter introduces means to reduce the amount of storage required in each name server for configuration data and the amount of update activity required to add new name servers or named objects to the environment.

3.3 Distributed Operations

3.3.1 Basic steps

Performing a name service operation on the attributes of an object involves first determining and locating authoritative name servers for the named object, and then accessing the appropriate attribute tuples. Specifically, these distributed operations consist of several steps:

1. Determine the authoritative name servers for the object;
2. Get the internet addresses of the authoritative servers;
3. Select the authorities necessary to perform the operation;
4. Perform the appropriate database operations at the machines on which the selected servers run;
5. Return the result, if any, to the calling client.

The first step, name resolution, uses the authority attributes stored in the configuration database. The second step requires additional configuration data, as described in the next section. The semantics of a particular name server operation are embodied in the last three steps. The third and fourth steps make use of the replicated data facilities to query or update the name service database. Note that the selection of authorities in step three depends strongly on the replication algorithm employed. The last step simply returns the result of the operation as specified in the name service interface.

3.3.2 Locating name servers

Name servers, like all other objects, may exist anywhere in the network and, hence, must be located before they can be accessed. The main attribute maintained about a name server is its internet address,

```
-- AttributeType = "InternetAddress" --
InternetAddressData: TYPE = NETADDRESS;
```

While name agents only need to discover the location of a single name server in order to utilize the name service, name servers should be able to locate other servers without resorting to global broadcast. Assume, for now, the number of servers is small enough that a database of server addresses can be feasibly stored at all servers:

```
serverDB: Database.DatabaseObject;
```

This database is part of the overall configuration database.

With a local database of server addresses, the procedure to locate servers is a simple database query:

[2] Mesa's **loophole** construct provides a way of subverting its strong type-checking. The first argument of the loophole is taken to be of the type given by the second argument.

```
LocateServers: PROCEDURE[servers: AuthorityList]
RETURNS[Replicated.StorageSites] = BEGIN
    address: NETADDRESS;
    sites: Replicated.StorageSites ← NIL;
    tuple: Database.AttributeTuple;
    WHILE servers # NIL DO
        tuple ← Database.Query[serverDB, servers.first,"InternetAddress"];
        address ← LOOPHOLE[tuple.value, NETADDRESS];
        sites ← CONS[address, sites][3];
        servers ← servers.rest;
        ENDLOOP;
    RETURN[sites];
    END;
```

The requirement that the name server address database be stored at all servers in its entirety will be relaxed in the next chapter.

3.3.3 Name service interface

Given the architecture for distributed name services developed in this chapter, all name servers present a common interface and accept lookup requests for any name from any client. Since the emphasis of this dissertation is not on designing a complete set of name service operations, two basic interface procedures, **Lookup** and **Update**, will suffice as sample operations in this and later chapters. Keep in mind, however, that a practical name service would likely desire a more sophisticated interface for reasons of performance and/or protection as discussed in Section 1.2.4.

Figure 3.4 presents a Mesa definitions module for the name service operations, which includes the type declarations for names and attributes. Assuming each name server has a single local database object,

```
localDB: Database.DatabaseObject;
```

a prototype implementation module might include:

```
Lookup: PROCEDURE[name: Name, attribute: AttributeType]
RETURNS[AttributeValue] = BEGIN
    authorities: AuthorityList;
    sites: Replicated.StorageSites;
    tuple: Database.AttributeTuple;
    authorities ← Resolve[name];
    sites ← LocateServers[authorities];
    tuple ← Replicated.Query[sites, localDB, name, attribute];
    RETURN[tuple.value];
    END;

Update: PROCEDURE[op: UpdateOps, name: Name, attribute: AttributeType,
value: AttributeValue] RETURNS[] = BEGIN
    authorities: AuthorityList;
    sites: Replicated.StorageSites;
    authorities ← Resolve[name];
```

[3] The cons constructor, in this procedure, adds an element to the beginning of a list. This facility is part of Cedar's extensions to Mesa.

```
NS: DEFINITIONS = BEGIN

Name: TYPE = STRING;
AttributeType: TYPE = STRING;
AttributeValue: TYPE = STRING;

Lookup: PROCEDURE[name: Name, attribute: AttributeType]
RETURNS[AttributeValue];

UpdateOps: TYPE = {add, delete, modify};

Update: PROCEDURE[op: UpdateOps, name: Name, attribute:
AttributeType, value: AttributeValue] RETURNS[];

END.
```

Figure 3.4: **Name Service interface.**

```
    sites ← LocateServers[authorities];
    SELECT op FROM
        add =>
         Replicated.AddTuple[sites, localDB, [name,attribute,value]];
        delete =>
         Replicated.DeleteTuple[sites, localDB, [name,attribute,value]];
        modify =>
         Replicated.ModifyTuple[sites, localDB, [name,attribute,value]];
        ENDCASE ;
    END;
```

The five steps outlined previously are represented in these **Lookup** and **Update** implementations. Notice that all external communication is encapsulated in the replicated data facilities.

A client's name agent might present a procedure call interface identical to that of the name service, as in Figure 3.5. A simple name agent of this sort could merely use the hypothetical remote procedure call mechanism to invoke name service operations:

```
mainServerAddress: NETADDRESS;

Lookup: PROCEDURE[name: Name, attribute: AttributeType]
RETURNS[AttributeValue] = BEGIN
    value: AttributeValue;
    value ← NS.Lookup[name, attribute] AT mainServerAddress;
    RETURN[value];
    END;

Update: PROCEDURE[op: UpdateOps, name: Name, attribute:
AttributeType, value: AttributeValue] RETURNS[] = BEGIN
```

```
NA: DEFINITIONS IMPORTS NS = BEGIN

Name: TYPE = NS.Name;
AttributeType: TYPE = NS.AttributeType;
AttributeValue: TYPE = NS.AttributeValue;

Lookup: PROCEDURE[name: Name, attribute: AttributeType]
RETURNS[AttributeValue];

UpdateOps: TYPE = NS.UpdateOps;

Update: PROCEDURE[op: UpdateOps, name: Name, attribute:
AttributeType, value: AttributeValue] RETURNS[];

END.
```

Figure 3.5: **Name Agent interface.**

```
    NS.Update[op, name, attribute, value] AT mainServerAddress;
    END;
```

The address of the name server to send requests to must be obtained by means other than the name service, such as broadcast probes sent over a local network [Boggs 83].

3.4 Summary

A distributed name service is provided by a collection of name servers that rely upon existing facilities for communication and database management to manage a name space in a decentralized fashion. This chapter presented an architecture for a distributed name service that allows the authority for parts of the name space to be freely divided amongst the various organizations participating in the distributed computing environment. The major difference between centralized and decentralized name management is the need to resolve names when the name space is dispersed throughout the environment.

Chapter 4

Name Resolution

Structure-free name resolution, unlike existing naming mechanisms, locates the set of authoritative servers for a named object without relying on the structure of the name space. Names are clustered, not necessarily syntactically, into contexts according to space and performance considerations. Name resolution proceeds by a series of context bindings until it encounters an authorities attribute for the named object. Structure-free name resolution permits easy reconfiguration of the service since an object's name remains independent from the location of its attributes or the details of its resolution. The amount of configuration data maintained by a name server can be easily reduced by lengthening the resolution chain for object names. Different styles of resolution allow the mechanism to be tailored to the division of computational power between servers and clients, as well as to the available communication paradigms.

4.1 Name Resolution Model

4.1.1 Distributing configuration data

Name resolution denotes the process of determining the authoritative name servers for a named object. In the name service architecture developed in this dissertation, the authorities for a named object are stored as the value of an "Authorities" attribute tuple. The previous chapter presented a simple model of name resolution in which the set of authorities attributes for every object, constituting the *configuration database*, was maintained in its entirety at every name server. Thus, all names could be resolved in a single step by any name server.

For environments with large numbers of objects, the configuration database may likely be too large to be stored everywhere. The knowledge of authorities for various named objects must be distributed so that no server needs complete knowledge of the configuration. The primary difficulty in resolving a name then lies in locating the authority attribute tuple for an object. Several interactions between servers may be required as the name resolution activity migrates from one name server to a potentially more knowledgeable server until the set of authoritative servers is determined.

4.1.2 Context objects

For the purpose of name resolution, *contexts* provide a means of partitioning the configuration database so that it may be distributed among servers. Contexts represent indivisible units for storage and replication of configuration database tuples. A context is thus materialized as an object containing configuration data.

```
ContextObject: TYPE = LIST OF ConfigTuple;
```

for now, assume that tuples holding configuration data are identical to other database tuples:

```
ConfigTuple: TYPE = Database.AttributeTuple;
```

This notion will be slightly modified in the next section.

Contexts have names just like any other object known to the name service,

```
ContextName: TYPE = Name;
```

and may be maintained at any collection of name servers, listed in an "Authorities" configuration tuple. However, contexts differ from other objects registered with the name service in that they are actually managed by the name service and central to its functioning. Also, The choice of particular names for contexts is not important since context names are only used internally within the name service.

Since the configuration database, stored in context objects, contains no attributes for clients' objects, its distribution should be of no concern to clients of the name service. Thus, the decomposition of the configuration database into context objects and the choice of authorities for those contexts can be done to facilitate name resolution, rather than being governed by administrative desires. The next section presents criteria for this decomposition.

4.1.3 Clustering conditions for configuration tuples

A *clustering condition* is an expression that allows the name space to be conveniently partitioned into contexts. Specifically, a clustering condition applied to a name yields either a TRUE or FALSE value:

```
ClusteringProc: TYPE = PROCEDURE[name: Name] RETURNS[BOOLEAN];
```

Any procedure that exhibits this behavior might serve as a clustering condition. The particular value returned, TRUE or FALSE , indicates whether or not the given name exists in the particular cluster.

Names can be clustered *algorithmically* according to the value that results from applying a function to them. In this case, the clustering condition is of the form "f[name] = value". For instance, a hash function is a well-known technique for clustering names into buckets.

More typically, clustering is done *syntactically* through pattern matching. *Patterns* are templates against which a name is compared. They range from names that may simply contain wildcards, which are denoted by "*" and match any sequence of characters, to regular expressions. Names matching a particular pattern, such as names with a common prefix "prefix.*", are considered part of the same cluster. That is, the clustering condition, when applied to a name, returns TRUE if the name matches the pattern.

Recent work on attribute-based naming conventions suggests a third type of clustering condition: *attribute clustering*. In this case, names are grouped according to what attributes they possess. For instance, an attribute-based name might consist of an unordered set of attribute type/value pairs of the form "AttributeType = AttributeValue" [IFIP 84]. Each attribute of this form could serve as a clustering condition; all names containing a particular attribute type with a particular value, such as "Organization=U.C.Berkeley", would belong to the same cluster.

Clustering conditions are used to assign names to contexts. That is, the authority attributes for all names belonging to a given cluster are stored in a single context object. Section 3.2.2 portrayed a situation in which all names exist in a single context that is stored at all servers. Clustering conditions may be applied to an existing context to further partition the context into smaller contexts.

Often, configuration attributes apply to a cluster of named objects. For instance, the names belonging to a given context might all have the same authoritative name servers. Thus, configuration attribute tuples are redefined to contain clustering conditions instead of fully qualified names:

```
ConfigTuple: TYPE = RECORD[
    cluster: ClusteringProc,
    attribute: NS.AttributeType,
    value: NS.AttributeValue
    ];
```

Configuration tuples resemble ordinary database tuples, except they can be considered attributes for all names satisfying the clustering condition. Note that a configuration tuple for a specific named object could contain a degenerate syntactic clustering condition that matches only the particular name.

4.1.4 Context bindings and name resolution chains

Once the configuration database is partitioned into various contexts, the process of name resolution is no longer a simple database query. When presented with a name to be resolved, a server might first look in local contexts for an authority attribute for the named object; if the authority can not be readily determined, additional configuration data must exist locally that enables the server to direct the resolution to another context, perhaps on a different server.

Context bindings, bindings between names that exist in a context and information that allows name resolution to proceed, direct the name resolution activity based on clustering conditions. The server trying to resolve a name applies a series of clustering conditions to the name until one of them is satisfied. Associated with each clustering condition is the name of another context in which to look for authority attributes of names in the cluster. This information is maintained in configuration tuples of type "ContextBinding":

```
-- AttributeType = "ContextBinding" --
ContextBindingData: TYPE = RECORD[
    newContext: ContextName
    ];
```

Contexts may contain configuration tuples of types "Authorities" and "ContextBinding".

Specifically, the algorithm for resolving names works as follows: Given a name to be resolved in some context, the particular context is searched for either an authorities attribute for the named object or a context binding containing a clustering condition that yields TRUE when applied to the name; in the latter case, the name is then resolved in the new context specified by the context binding attribute. Thus, resolving a name is a matter of successively binding names within contexts until the authoritative name servers for the named object are discovered. That is, the name resolution mechanism traverses a *resolution chain* of "ContextBinding" attribute tuples until it encounters an "Authorities" attribute.

When a name is originally presented for resolution, an *initial context* must be chosen in which to start the resolution chain:

```
initialContext: ContextObject;
```

The initial context must contain authority attributes or context bindings for all names in the name space.

Global names result if and only if the initial context is a global one, that is, all name servers share a common initial context. Relative names arise if the initial context used in name resolution is not a global one, but is relative to the particular server presented with the resolution request or to some other implicit context. The UUCP network for sending mail presents a good example of a relative naming convention arising from interpreting recipient names relative to the sender's machine [Nowitz 78].

4.1.5 Applying the name resolution model

At this point, examples of how the model of name resolution presented above can be used to describe existing naming conventions should help to clarify matters. The set of clustering conditions chosen by a given naming system partitions the name space such that each name exists in exactly one cluster; each cluster is stored in a separate context object. Generally, existing naming conventions can be characterized by the types of clustering used.

4.1.5.1 Syntactic clustering

Syntactic clustering allows names to be resolved in a manner similar to their structure, as is done by virtually all current name management systems; simple pattern matching suffices as a clustering technique. That is, suppose a routine exists that takes a name and a pattern as arguments and returns an indication of whether the name matches the pattern:

```
Pattern: TYPE = STRING;

Matches: PROCEDURE[name: Name; pattern: Pattern] RETURNS[BOOLEAN];
```

Current approaches to name management rely solely on clustering procedures consisting of a single pattern match:

```
PatternCP: ClusteringProc = BEGIN
    RETURN[Matches[name, "some-pattern"]];
    END;
```

The particular approaches can be classified according to the name structure's effect on name resolution:

Authority-dependent names: Names with the structure, "subname.server", explicitly indicate the authorities over parts of the name space. Technically, such a scheme requires no configuration data. Conceptually, a virtual context exists with an attribute for each server,

[Matches[name,"*.server"],"Authorities","server"] .

A name space of this sort is said to be *physically partitioned* since a name reveals the physical storage site of information about its referent.

Organizationally partitioned names: A name space that is *organizationally partitioned*, as used by Grapevine [Birrell *et al.* 82], allows flexible name management since the organizational authority for assigning names is explicitly recognized, but decoupled from the authoritative name servers for those names. With such a naming scheme, the database partitions correspond to organizations rather than name servers. With names of the form, "subname.org", the initial context contains a context binding tuple for each organization,

[Matches[name,"*.org"],"ContextBinding","org"]

while each organization maintains a context object containing authority attributes for all named objects within that organization. An organization's name serves as a convenient name for its context.

In Grapevine, all named objects within an organization have identical authorities, so each organization's context contains a single attribute,

[Matches[name,"*.org"],"Authorities","$server_1, \ldots, server_K$"]

A more general name distribution scheme requires an authorities attribute for each named object. An organization's context object would be of the form:

org:
[Matches[name,"$name_1$.org"],"Authorities","$server_{11}, \ldots, server_{1K}$"]
.
.
.
[Matches[name,"$name_N$.org"],"Authorities","$server_{N1}, \ldots, server_{NK}$"]

assuming the organization contained N named objects that had K authoritative servers each.

Hierarchical names: Organizations can themselves be partitioned into smaller clusters, resulting in *hierarchical* names consisting of more than two parts. The contexts at the lowest level of the hierarchy contain the authority attribute tuples, while those at higher levels contain context bindings, which indicate a delegation of authority for managing parts of the name space. The amount of configuration data that must be stored in context objects at the various levels of the hierarchy is proportional to the degree of branching of the name space tree. For this reason, hierarchical naming conventions with several levels are often well suited for naming large numbers of objects.

Consider the name space depicted structurally in Figure 4.1. The inherent structure in the name space can be exploited by applying syntactic clustering conditions as indicated in Figure 4.2. In the example, names are initially clustered according to their last character. Clusters that are too large to be conveniently stored as a single context, perhaps the set of names ending in "A" in Figure 4.2, can be further partitioned by applying additional clustering conditions. Figure 4.3 presents a complete configuration database needed to resolve these names.

4.1.5.2 Variable syntactic clustering

Although existing name management mechanisms for hierarchical name spaces resolve names a label at a time, as is done in Figure 4.2, syntactic clustering conditions are not restricted to matching a single additional label in each step. That is, even using syntactic clustering, the length of the resolution chain for various names need not correspond exactly to the number of labels in the names. Name resolution can be tailored according to the desired response time for resolving names and the size of contexts.

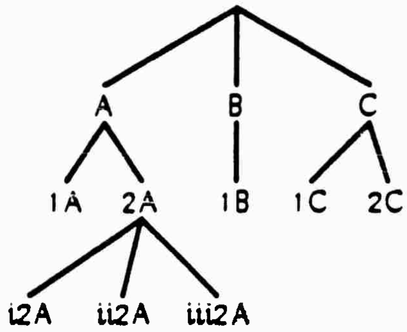


Figure 4.1: Sample hierarchical name space.

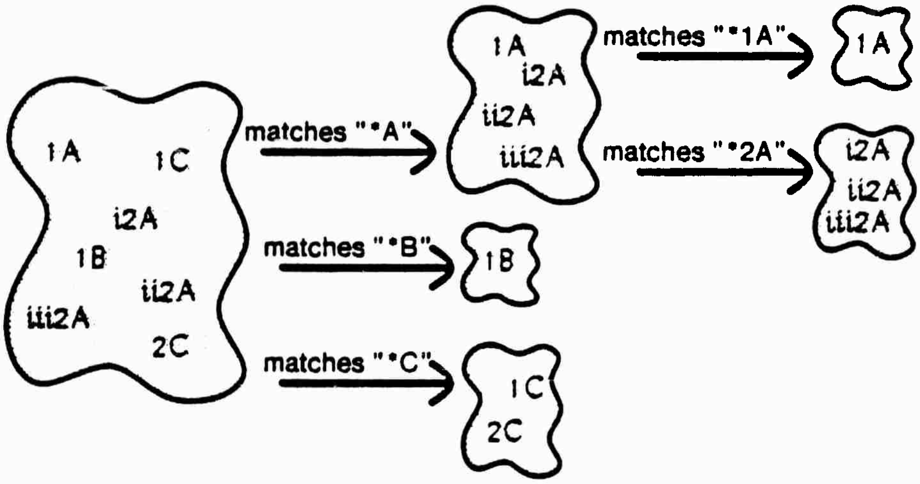


Figure 4.2: Syntactic clustering of a hierarchical name space.

```
Initial:
        [Matches[name,“*A”],“ContextBinding”,“A”]
        [Matches[name,“*B”],“ContextBinding”,“B”]
        [Matches[name,“*C”],“ContextBinding”,“C”]
A:
        [Matches[name,“*1A”],“ContextBinding”,“1A”]
        [Matches[name,“*2A”],“ContextBinding”,“2A”]
1A:
        [Matches[name,“1A”],“Authorities”,“···”]
2A:
        [Matches[name,“i2A”],“Authorities”,“···”]
        [Matches[name,“ii2A”],“Authorities”,“···”]
        [Matches[name,“iii2A”],“Authorities”,“···”]
B:
        [Matches[name,“1B”],“Authorities”,“···”]
C:
        [Matches[name,“1C”],“Authorities”,“···”]
        [Matches[name,“2C”],“Authorities”,“···”]
```

Figure 4.3: **Configuration database for syntactic clustering.**

Once again, consider the name space in Figure 4.1. Suppose the initial context has enough storage space to contain four context bindings instead of three. All names, including those with three labels, can be resolved in a single step by matching multiple labels at time as demonstrated in Figure 4.4. If the name space grows over time, then the intermediary context binding present in Figure 4.2 can be easily reintroduced; no names need to change, only their clustering and distribution.

Syntactic clustering in which a variable number of labels can be matched allows potential performance advantages to be obtained over traditional resolution schemes. Particularly, regions of the name space that are abnormally sparse may be clustered together for purposes of name resolution. Also, the name resolution chain for special names can be reduced by adding new clustering conditions that match larger components of the names than a single label. Section 4.1.7 formalizes these space/time tradeoffs.

4.1.5.3 Non-syntactic clustering

Algorithmic clustering allows names to be resolved independent of their structure. Clients of the name service can choose names for their objects without requiring agreed-upon name structures; the only requirement is that names be unambiguous. Hashing represents a familiar way of clustering names algorithmically.

Suppose functions exist that map a name into a real number in the range (0..1], such as,

```
Hash: PROCEDURE[name: Name] RETURNS[REAL];
```

The name space of Figure 4.1 could be partitioned as in Figure 4.5. Notice that the partitions do not correspond to the inherent structure of the name space. In fact, the name resolution tree is binary while the name space has various branching factors if one looks at it syntactically.

A complete configuration database for these names is given in Figure 4.6. Pattern matching is used for authority attributes since each object has its own set of authoritative name servers. Both the

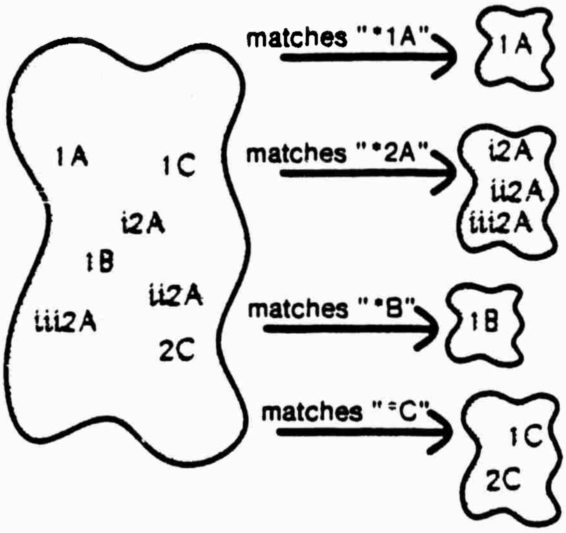


Figure 4.4: Clustering varying numbers of labels.

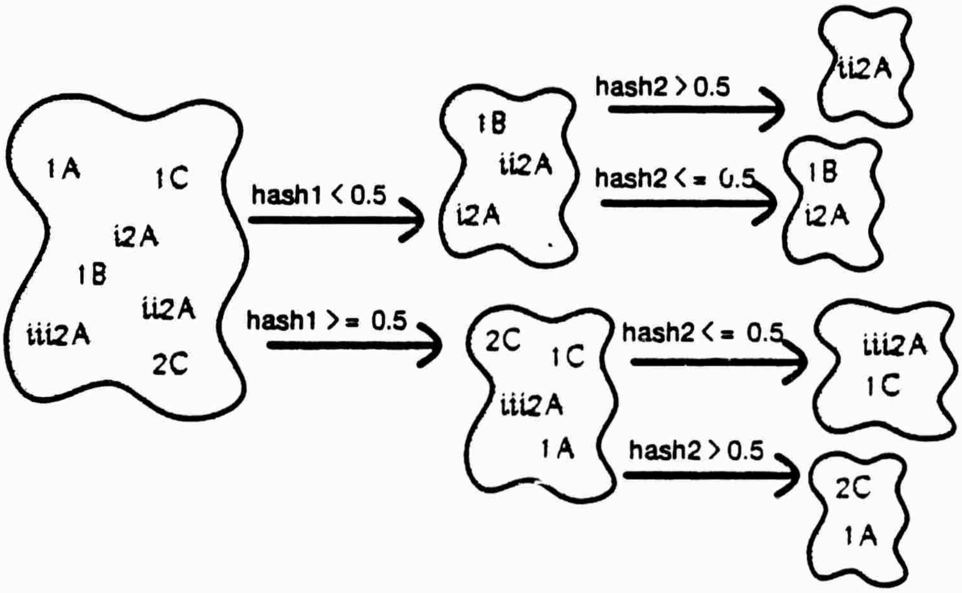


Figure 4.5: Clustering a name space through hashing.

```
Initial:
        [Hash1[name]< 0.5,"ContextBinding","C1"]
        [Hash1[name]>= 0.5,"ContextBinding","C2"]
C1:
        [Hash2[name]> 0.5,"ContextBinding","C3"]
        [Hash2[name]<= 0.5,"ContextBinding","C4"]
C2:
        [Hash2[name]<= 0.5,"ContextBinding","C5"]
        [Hash2[name]> 0.5,"ContextBinding","C6"]
C3:
        [Matches[name,"ii2A"],"Authorities","···"]
C4:
        [Matches[name,"1B"],"Authorities","···"]
        [Matches[name,"i2A"],"Authorities","···"]
C5:
        [Matches[name,"iii2A"],"Authorities","···"]
        [Matches[name,"1C"],"Authorities","···"]
C6:
        [Matches[name,"2C"],"Authorities","···"]
        [Matches[name,"1A"],"Authorities","···"]
```

Figure 4.6: Configuration database for algorithmic clustering.

configuration database in Figure 4.3 and the one in Figure 4.6 allow the set of names to be resolved, but in drastically different ways. Even the name resolution chains for a given name vary in length for the different clustering strategies.

4.1.5.4 Mixed clustering for growing systems

A mixture of syntactic and non-syntactic clustering can often prove useful for resolving names in evolving systems. Current problems of scale in the Grapevine system serve as a good example. Grapevine clusters names syntactically based on the registry name embedded in all object names. Some of Grapevine's registries are becoming quite large. Suppose that a particular registry grows too large to be feasibly managed as a single context; what can be done?

One course of action might be to add another layer to the name structure, yielding three-part names as was done for the clearinghouse system. Unfortunately, this approach forces all objects to change their names, a costly operation for well-established systems. It also requires changes to Grapevine's name resolution mechanism. Within the framework of the existing Grapevine system, the only solution is to split the registry into two separate registries. Again, some or all members of the registry must change their names.

A better approach might be to algorithmically partition large registries into smaller clusters. The resolution chains for some object names would grow from one link to two; the first context binding being done syntactically, while the second is done perhaps by a hash function as depicted in Figure 4.7. Thus, changes to the Grapevine servers' resolution mechanism are required, but no object names need to change.

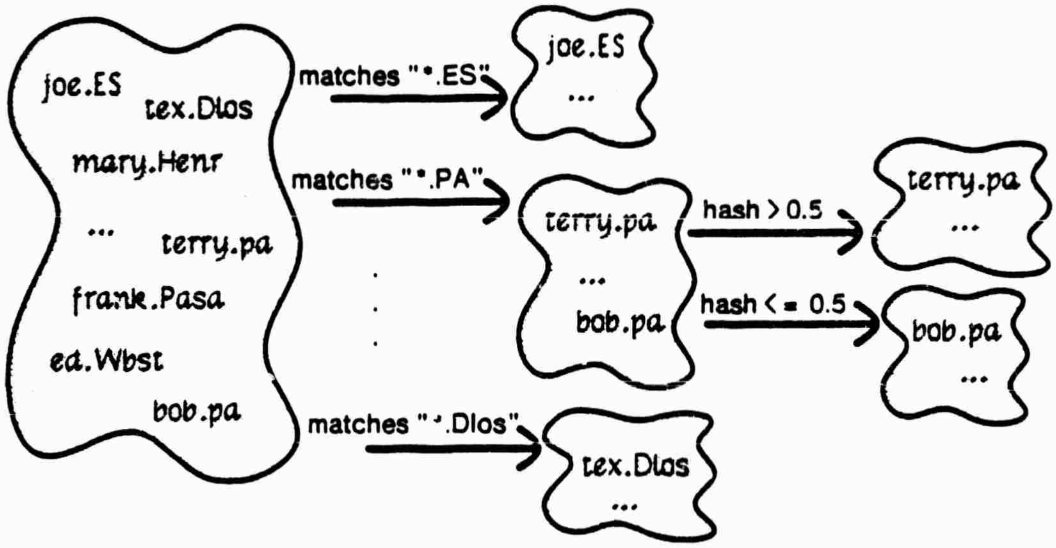


Figure 4.7: **Clustering large Grapevine registries algorithmically.**

4.1.6 Extensions for other naming styles

In all of the name management schemes described thus far, the name to be resolved at any point in the resolution chain did not change; only the context in which to resolve the name changed. Some naming mechanisms involve changing the name being resolved as well as the context. Often, this new name is a function of the old name to be resolved, perhaps some partially qualified part of the old name:

```
PartialName: TYPE = Name;
NewNameProc: TYPE = PROCEDURE[pname: PartialName]
    RETURNS[PartialName];
```

To support these more elaborate styles of naming, context binding configuration attributes must be extended to include the new name to be resolved in the new context:

```
ContextBindingData: TYPE = RECORD[
    newContext: ContextName,
    newName: NewNameProc
    ];
```

In all of the previously described conventions, the NewNameProc was simply the identity mapping. However, it could also have been a name reduction mapping in which the new name is a strict tail component of the old name. Such name reductions can either be used solely to reduce the amount of storage required in context objects or to guarantee termination of the name resolution chain.

4.1.6.1 Naming networks

Hierarchical naming conventions are special cases of the more general *naming networks* in which objects are identified by *path names* [Saltzer 78]. Naming networks can be easily built up from the name resolution model presented because of the general relations allowed between contexts via

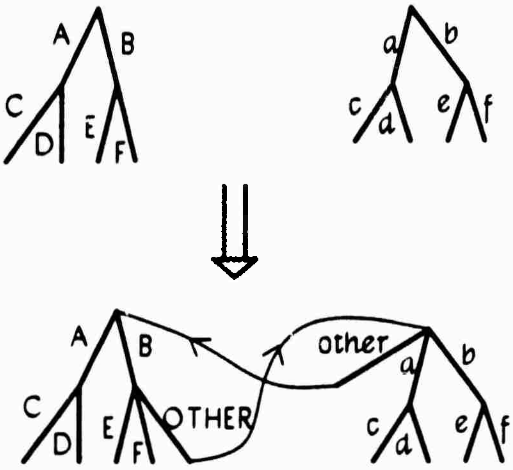


Figure 4.8: **Mutually encapsulated name spaces.**

context bindings. In a naming network, names are resolved syntactically a label at a time as in hierarchical name spaces, but cycles may exist in a name resolution chain. Because of these cycles, name truncation is necessary to halt the resolution. That is, the new name to be resolved is derived from the old name by stripping off a label; the name resolution process terminates when only a single label remains.

Naming mechanisms in which the name left to be resolved at any point in the resolution process is a tail component of the original name presented for resolution are defined herein as *predestinate* naming conventions. Naming networks typify the class of predestinate naming conventions. Notice that, for naming conventions of this sort, a name strictly decreases in length as the resolution proceeds, thus ensuring that the resolution activity will eventually terminate. For example, consider the name "A.B.C.D.E" complying with a hierarchical naming convention or naming network. The resolution chain is as follows:

INITIAL(A.B.C.D.E)
⟶ A(B.C.D.E)
⟶ A.B(C.D.E)
⟶ A.B.C(D.E)
⟶ A.B.C.D(E)

The name resolution mechanism simply scans the name from left to right extracting a label at a time and migrating to an authority for the new context obtained by concatenating the label just scanned with the previous context name.

Naming networks that are not strictly hierarchical might naturally arise in practice when two existing hierarchical name spaces wish to reference each other's objects by mutually encapsulating their name spaces, as depicted in Figure 4.8. Clients of the first name space can reference objects in the second by prepending their names with "B.OTHER.", whereas clients of the second name space can reference objects in the first by prepending their names with "other.". Notice, however, that the two name spaces retain their original separate initial contexts, probably for backward compatibility. In this example, the naming network resulting from the junction of the two original name spaces is not only unrooted, but also has cycles.

4.1.6.2 Beyond naming networks

Name resolution is not limited to predestinate naming conventions, such as naming networks, ıor which the resolution chain is predictable from the syntax of the name. In particular, within the name resolution configuration data, the newName and newContext fields of a "ContextBinding" attribute need bear no relationship to the containing context's name or the current name being resolved, or the relationship may not be as simple as stripping off a single label of the name.

As a simple example of *non-predestinate* name resolution, consider the convention for naming Arpanet mail recipients currently used within the U. C. Berkeley Computer Science Division. Mail clients are named according to the convention "user@Berkeley", though, internally, users are partitioned according to what computer they use. The name "frank" might exist in the context of machine "ernie", while "joe" exists on machine "kim"; though their official mail addresses are "frank@Berkeley" and "joe@Berkeley", respectively. Thus, the "Berkeley" context might contain two context bindings for these users as follows:

Berkeley:
[Matches[name,"frank"],"ContextBinding","Berkeley.ernie(frank)"]
[Matches[name,"joe"],"ContextBinding","Berkeley.kim(joe)"]

In these cases, the context bindings discard no components of the name to be resolved; only the context itself becomes more refined.

Subaliases, aliases for particular components of a name, fit nicely into the context binding model. For instance, if "Berkeley" is a subalias for "ucbvax", the two names can be made interchangeable by a NewNameProc that takes a name of the form "front.Berkeley.back" and returns "front.ucbvax.back".

In general, *mapping contexts* that change a name to be resolved in a wide context to a new name in some smaller context are useful for converting between standard global names and naming conventions particular to the internals of an organization. The inevitable evolution of distributed computing environments often makes name conversions between old and new formats necessary. The rewriting rules incorporated into the *Sendmail* internetwork mail router [Allman 83] were a response to conversion requirements between various existing mail facilities. The pattern matching abilities in context objects and the generality in context bindings allow them to accommodate such conversions within the name resolution architecture.

4.1.7 Advantages of structure-free name resolution

The model of name resolution developed in this dissertation in which the process of resolving names need not be strictly tied to the name structure, *structure-free name resolution*, permits names to be managed more flexibly than existing naming mechanisms. Specifically, it allows tradeoffs to be made in how names are managed without affecting the structure of the names or the resolvability of the names. These space/time tradeoffs are demonstrated by the following two rules, which change the content and distribution of the configuration database while preserving name resolution:

The partition rule: *Let DB be a context and c be a clustering condition applied to names in that context; if all names in DB for which the clustering condition c applied to them yields true are removed from DB and placed in a new context DB_c, and one attribute tuple is added to DB:*

[c[name],"ContextBinding","DB_c"]

then all names that could be resolved in the old DB context can be resolved in the new one.

The indirection rule: *Let DB be a context whose authorities are $A_1, A_2, \ldots, A_n$ where $n >= 2$; if name server A_1 replaces its local context DB with a new context $DBnew$ containing two attribute tuples:*

[Matches[name,"*"],"ContextBinding","DB"]
[Matches[name,"DB"],"Authorities","$A_2, \ldots, A_n$"]

then all names that could be resolved in the DB context can be resolved in the $DBnew$ context.

Theses two rules govern the modifications that can be made to reduce the overall amount of configuration data without impairing name resolution. In particular, the partition rule provides a way of splitting up a large context into smaller, more manageable, pieces; the indirection rule allows a name server to offload the responsibility for maintaining a context to other servers, thereby reducing its local storage requirements. Note that both rules add another binding to the resolution chain for certain names, thus increasing the time to resolve a name. On the other hand, the indirection rule reduces the total amount of storage required in the name service, assuming contexts are larger than a couple of attribute tuples.

Starting with a single context stored at all name servers that contains the complete set of "Authorities" attributes for all named objects, these two rules can be repeatedly applied to partition and distribute the configuration data while ensuring that all names can be resolved. The rules are not meant to represent operations that can be performed on a running system. Rather, they suggest the range of options available to administrators when configuring or reconfiguring a name service.

Importantly, the cost of name resolution varies with the amount of storage dedicated to configuration data. At one extreme, if all servers have enough storage to hold the complete set of authority attributes for all named objects, then any name can be resolved in a single step. On the other hand, if authority attributes are distributed among servers, then context bindings are needed and name resolution becomes more costly. Chapter 5 quantifies how the cost of resolution varies with the length of the resolution chain.

Different name servers may observe different costs for name resolution depending on how much configuration data they store locally. One small name server with very little storage need not increase the name resolution chains for the complete service; only the particular server's clients are affected. If certain name servers are upgraded with additional storage, gains in name resolution can be achieved for some names.

4.2 Name Resolution Mechanism

4.2.1 Configuration database queries

The name service configuration database consists of a collection of contexts that are stored and replicated on various name servers. Looking up a name in a context involves applying a configuration attribute's clustering condition to the name until one that returns TRUE is discovered. This is performed by the **Query** operation of the **Cluster** module[1]:

```
-- record format for storing ConfigTuples in AttributeTuples
CTuple: TYPE = RECORD[
     unused: NS.Name,
     attribute: NS.AttributeType,
     cluster: ClusteringProc,
     value: NS.AttributeValue
     ];
```

[1] For simplicity, all exceptional condition handling is left out of the prototype implementation. Particularly, this procedure assumes that, for any name, some clustering conditioning in the context yields **true**. This can be easily ensured by ending every context with a clustering condition that always returns **true**.

```
Query: PROCEDURE[cname: ContextName, name: Namc]
RETURNS[ConfigTuple] = BEGIN
    tuple: Database.AttributeTuple;
    next: Database.TupleID ← NIL;
    dbContext: Database.DatabaseObject;
    configData: ConfigTuple;
    cb: CTuple;
    dbContext ← ContextNameToObject[cname];
    DO
        [tuple, next] ← Database.Enumerate[dbContext, next];
        cb ← LOOPHOLE[tuple, CTuple];
        IF cb.cluster[name] = TRUE THEN EXIT;
        ENDLOOP;
    configData ← [cb.cluster,cb.attribute,cb.value];
    RETURN[configData];
    END;
```

This routine uses the ordinary database facilities to store configuration data tuples. Names of locally stored contexts are mapped to the appropriate database object by the **ContextNameToObject** routine.

The routine for querying configuration attributes makes use of the single-site database facilities rather than the replicated data facilities. Since configuration data changes infrequently and name resolution should proceed as quickly as possible, fancy techniques for replicated context objects are unwarranted. The name resolution algorithm that calls upon the **Cluster** module assumes that all copies of a context are up-to-date and chooses one to suit its needs. This allows different styles of resolution to be accommodated as demonstrated in Section 4.2.3.

4.2.2 Locating context objects

Since names are always resolved with some context, a major problem in resolving names is determining the authoritative servers for the particular context. Contexts are themselves objects that may be distributed and replicated in any number of name servers. Thus, as with other types of objects, locating a context involves resolving its name,

```
FindContext: PROCEDURE[cname: ContextName]
RETURNS[AuthoritiesData] = BECIN
    RETURN[Resolve["initialContext", cname]];
    END;
```

However, the attempt to locate the context was triggered by the process of resolving a name in the first place. Thus, if the FindContext routine calls Resolve, infinite recursion results unless some special cases are utilized for locating certain contexts. That is, some special way of locating contexts must be provided as the base case of the recursive name resolution.

One approach is to have a special "context" context containing the authoritative name servers for all other named contexts, also referred to as a *metacontext*. Locating a context, then, would simply involve binding that context's name in the special metacontext. The problem then becomes locating the metacontext. Fortunately, the metacontext is small compared to the complete name server database since it contains only information about contexts. Moreover, it changes very infrequently. Thus, in many cases, the metacontext can be stored at all name server sites, making it readily available for resolving context names:

```
myself: ServerName; -- name of local server

FindContext: PROCEDURE[cname: ContextName]
RETURNS[AuthoritiesData] = BEGIN
    IF cname = "metaContext" THEN
        RETURN[myself]
    ELSE
        RETURN[Resolve["metaContext", cname]];
    END;
```

For very large name spaces with many contexts, however, even the metacontext may consume more storage than some name servers can afford. In this case, such servers need only store references to the servers that actually store the metacontext and not the context itself by making use of the **indirection rule**. A remote metacontext contains the actual "Authorities" attributes for all contexts, while the local metacontext needs only two tuples:

```
metaContext:
    [Matches[name,"*"],"ContextBinding","remoteMetaContext"]
    [Matches[name,"remoteMetaContext"],"Authorities","···"]
```

Context names can be easily resolved by calling on an authoritative server for the remote metacontext. The servers for the metacontext can be viewed as providing a special name service for context objects.

For widely distributed name spaces, a better approach to requiring the existence of a metacontext is to distribute the context configuration database just like the configuration data for other objects is decentralized. In order to guarantee that a name server can resolve any context name presented to it without contacting other servers, a context that contains context bindings to other contexts should also include the authority attributes for those contexts. With this coupling of context bindings and authority data, a new context name can always be readily resolved in the current context:

```
FindContext: PROCEDURE[oldContext: ContextName, newCname:
ContextName] RETURNS[AuthoritiesData] = BEGIN
    RETURN[Resolve[oldContext,newCname]];
    END;
```

Context names appearing in a context binding, rather than being globally unambiguous, are thus relative to the context in which the context biriding occurs. Without a single metacontext, no context must grow with the size of the complete name space. Each name server need only maintain knowledge of a localized portion of the name space.

4.2.3 Styles of name resolution

4.2.3.1 Recursive

While the policy for resolving names according to a particular naming convention is embodied in the contents of context objects, the mechanics of name resolution is independent of the given adopted naming convention. One algorithm for resolving a name relative to a context is as follows:

```
Resolve: PROCEDURE[context: ContextName, name: Name]
RETURNS[AuthoritiesData] = BEGIN
    -- local variables
```

```
authorities, contextAuthorities: AuthoritiesData;
contextServer: ServerName;
contextAddress: Internet.Address;
binding: ContextBindingData;
tuple: ConfigTuple;
-- lookup name in context
tuple ← Cluster.Query[context, name];
SELECT tuple.attribute FROM
    "Authorities" =>
        authorities ← LOOPHOLE[tuple.value, AuthoritiesData];
    "ContextBinding" => BEGIN
        binding ← LOOPHOLE[tuple.value, ContextBindingData];
        contextAuthorities ← FindContext[context, binding.newContext];
        contextServer ← SelectServer[contextAuthorities];
        IF contextServer = myself THEN
            authorities ← Resolve[binding.newContext,
                binding.newName[name]]
        ELSE BEGIN
            contextAddress ← LocateServer[contextServer];
            authorities ← Resolve[binding.newContext,
                binding.newName[name]]
                    AT contextAddress;
            END;
        END;
    ENDCASE ;
RETURN[authorities];
END;
```

This algorithm is a recursive one in that names are recursively resolved in new contexts until an authoritative name server is determined. The name resolution activity migrates to servers containing the necessary contexts through remote procedure calls.

In the resolution algorithm presented above, the responsibility for performing the name service operation rests with the initial name server that received the operation request. This server returns the appropriate response after the name has been resolved and the operation performed. Using such a *recursive* style of name resolution, the name service appears to a name agent to be a centralized service; name agents may be unaware of the existence of multiple servers. However, because of the recursive nature of the name resolution mechanism, a disparity in work results: the name agent has little work to do while name servers may be involved in processing several requests at the same time. This disparity is particularly alarming when one realizes that an order of magnitude more name agents exist than name servers.

4.2.3.2 Iterative

An alternative to resolving names recursively is to use *iterative* name resolution in which the name agent retains control over the resolution activity. The algorithms are similar, except that servers do not call each other directly in the iterative case. A name server does its best to resolve names using only locally available configuration data and returns to the calling name agent when it can no longer continue. The name agent then calls on a different name server to continue resolution of the name.

A name service operation can be in one of two stages when a server replies to the calling name agent:

Unresolved. The name has been only partially resolved.

Resolved. The name has been completely resolved and the operation has been completed.

The state of the resolution process at any point in time can be represented by a context name and a name to be resolved in that context:

```
ResolveState: TYPE = RECORD[
    context: ContextName,
    name: Name
    ];
```

Initially, the resolution state consists of the initial context and a complete name.

When a server can not further resolve a name, it returns the current state along with the internet address of a server that the name agent should contact next, presumably an authority for the current context in the resolution chain[2]. The iterative version of the **Resolve** routine is thus as follows:

```
ResolveI: PROCEDURE[context: ContextName, name: Name]
RETURNS[ConfigTuple, NETADDRESS] = BEGIN
    tuple: ConfigTuple;
    address: NETADDRESS;
    binding: ContextBindingData;
    authorities: AuthoritiesData;
    server: ServerName;
    tuple ← Cluster.Query[context, name];
    IF tuple.attribute = "ContextBinding" THEN
        BEGIN
        binding ← LOOPHOLE[tuple.value, ContextBindingData];
        authorities ← FindContext[context, binding.newContext];
        server ← SelectServer[authorities];
        IF server = myself THEN
            [tuple, address] ← ResolveI[binding.newContext,
                binding.newName[name]]
        ELSE
            address ← LocateServer[server];
        END;
    RETURN[tuple, address];
    END;
```

The name agent is responsible for presenting the current resolution state to a name server along with an operation request so that the resolution activity can continue where it left off.

In order to allow iterative name resolution, all name service operations should take the current resolution state as an additional parameter. These operations must also return an indication of the stage of the operation,

```
Stage: TYPE = {Unresolved,Resolved};
```

along with enough information for processing to continue at another name server. If the name has not been completely resolved, the operation returns the current state of the resolution and the address of the next server to contact,

[2] In some cases, it would be better for the server to return the list of authorities rather than choosing one and returning its address. A name agents that has knowledge about the existence and locations of servers would be able to select a server based on its own criteria rather than the name server's. For instance, the closest authority to the server is not necessarily the closest to the agent. Moreover, name agents could cache the authority information and use it to intelligently direct future operations. Caching is discussed in more detail in Chapter 7.

```
UnresolvedData: TYPE = RECORD[
     state: ResolveState,
     next: NETADDRESS
     ];
```

If the operation has been completed, then the name agent receives the desired result of the operation, as usual.

For instance, the name server lookup routine for iterative name resolution has a similar interface to that for a recursive style of resolution, except it accepts the resolution state as a parameter and returns the current operation stage:

```
LookupI: PROCEDURE[name: Name, attribute: AttributeType,
state: ResolveState] RETURNS[AttributeValue, Stage] = BEGIN
-- iterative version of the Name Server
     tuple: Database.AttributeTuple;
     ctuple: ConfigTuple;
     address: NETADDRESS;
     authorities: AuthoritiesData;
     sites: Replicated.StorageSites;
     binding: ContextBindingData;
     continue: UnresolvedData;
     st: Stage;
     value: AttributeValue;
     [ctuple, address] ← ResolveI[ResolveState.context, ResolveState.name];
     SELECT ctuple.attribute FROM
          "ContextBinding" => BEGIN
               binding ← LOOPHOLE[ctuple.value, ContextBindingData];
               continue.state.context ← binding.newContext;
               continue.state.name ← binding.NewNameProc[name];
               continue.next ← address;
               value ← LOOPHOLE[continue,AttributeValue];
               st ← Unresolved;
               END;
          "Authorities" => BEGIN
               -- same as for recursive name server
               authorities ← LOOPHOLE[ctuple.value, AuthoritiesData]];
               sites ← LocateServers[authorities];
               tuple ← Replicated.Query[sites, localDB, name, attribute];
               value ← tuple.value;
               st ← Resolved;
               END;
          ENDCASE => ERROR;
     RETURN[value,st];
     END;
```

The value returned by this routine depends upon the stage of the operation, as does the name agent's action:

```
Lookup: PROCEDURE[name: Name, attribute: AttributeType]
RETURNS[AttributeValue] = BEGIN
-- iterative version of the Name Agent
     value: AttributeValue:
```

```
st: Stage;
continue: UnresolvedData;
state: NS.ResolveState ← [name,"initialContext"];
address: NETADDRESS← mainServerAddress;
DO
    [value, st] ← NS.LookupI[name, attribute, state] AT address;
    SELECT st FROM
        Unresolved => BEGIN
            continue ← LOOPHOLE[value, UnresolvedData];
            state ← continue.state;
            address ← continue.address;
            END;
        Resolved => EXIT;
        ENDCASE => ERROR;
ENDLOOP;
RETURN[value];
END;
```

Notice that the name agent's interface presented to its clients remains the same regardless of the style of resolution employed.

4.2.3.3 Transitive

A third style of name resolution, *transitive* name resolution, falls somewhere between recursive and iterative resolution. With transitive name resolution, the name server currently processing an operation simply passes the operation to a server that can continue its processing. As in the recursive approach, name agents are not involved in the act of name resolution; and, like the iterative approach, a name server gives up its responsibility for performing an operation when it can no longer resolve the name locally.

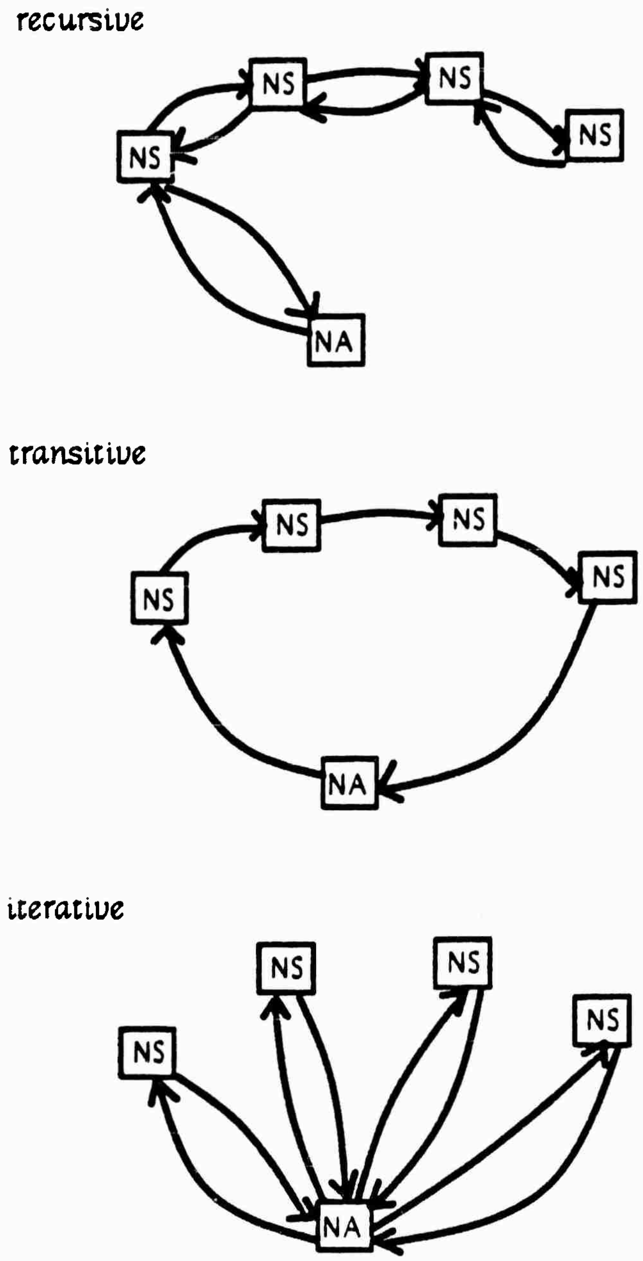
The implementation of transitive name resolution is similar to the iterative style presented above, except that the operation, along with its current state, is sent directly to the selected next server instead of returned to the name agent; the authoritative name server that eventually performs the desired operation returns the result. The only way that a name agent may be aware of the distributed nature of the service is that the response to its request may be received from a different server than the one it was sent to.

4.2.3.4 Comparisons

Figure 4.9 shows the communication patterns induced by the different styles of resolution. The choice of a particular style of name resolution should be based on the relative processing powers of name servers and name agents and on the semantics of the communication protocols employed.

The transitive approach to resolving names results in the fewest number of high-level messages, though it is more susceptible to failures since servers do not receive feedback once an operation is passed on. Thus, transitive resolution is best suited for an environment in which reliable communication connections between name servers can be cheaply maintained. Recursive and iterative styles of resolution, on the other hand, adapt nicely to a remote procedure call communication paradigm. An iterative approach also works well if an unreliable datagram protocol with timeouts is utilized; the periodic replies from servers makes it easy for the name agent to monitor and recover from failures. Timeouts are much harder to set with a recursive or transitive style because of the large variation in the time necessary to resolve names.

As for computation, the iterative style of name resolution requires the name agent to do more work. However, it also provides more opportunities for the name agent to play an intelligent role in name



recursive

transitive

iterative

Figure 4.9: Styles of name resolution.

resolution; for instance, a name agent may choose to cache the results of recent name resolutions and use this to direct future name lookups to the appropriate server. If an iterative approach is adopted, small computers with dumb name agents could be accommodated by interjecting a *name resolution server* between the agent and servers. Such a server would control the resolution activity so that the simple name agent need not be involved.

All three styles require approximately the same amount of computation from the name servers; the only major difference is the lack of communication between servers in the iterative case. Recursive resolution, however, would undoubtedly require the name server to be multiprogrammed since a server can not afford to wait for a recursive resolution request to return before processing the next request. Thus, the internal organization of a name server performing recursive name resolution is much more complicated than that for the other styles.

4.3 Dynamics of Name Management

4.3.1 Updates

Large distributed computing environments are constantly changing and evolving. Name services gain their utility by insulating users from the immediate effects of changes and allowing them to discover these changes through late binding. For instance, if an object moves then the name service should be informed of its new location; other objects that reference the moved object by name need not be aware of its migration since they locate it indirectly through the name service.

When designing a name service, one must allow updates to the name service database but try to isolate the effect of these updates, not only from name service clients, but also from as many servers as possible within the name service. The many kinds of updates to the name service include adding, removing, or changing:

- object attributes
- object names
- contexts
- authorities
- name servers

Changing the set of attributes for a given object, as previously discussed, requires first resolving the name and then performing a replicated database operation. Only the authoritative name servers for the object are involved. The other classes of updates are more difficult, and are discussed in detail in the next two sections.

4.3.2 Name registration

With the simple name service interface presented, registering or unregistering an object with the name service is simply a matter of adding an attribute for the object or removing all of the attribute tuples for the object, respectively. However, to guarantee that two different objects do not inadvertently register under the same name, rendering the name ambiguous, it may be desirable to provide additional name service routines:

```
Register: PROCEDURE[name: Name] RETURNS[];

UnRegister: PROCEDURE[name: Name] RETURNS[];
```

UnRegister is not strictly necessary since it simply deletes all of the attribute tuples associated with the named object; **Register**, on the other hand, has very special semantics.

Name service clients are allowed to choose a name for their objects, but a name must be determined to be unambiguous upon registration, that is, the name must not be already in use. The registration activity attempts to resolve the name presented for registration until either the resolution mechanism can no longer continue or an "Authorities" tuple for the name is encountered. In the latter case, the registration request is rejected since the name is already in use. In the former case, the part of the name that was to be resolved when the mechanism halted, the *remainder*, is added to the current context with a list of authorities. For example, if one attempts to register the name "A.B.C" and "A.B" is an existing context that contains no attributes for "C", then the name's resolution will fail when a "ContextBinding" or "Authorities" attribute for "C" is searched for in the context "A.B" and not found. At that point, an authorities tuple for "C" will be added to context "A.B".

It may be desirable to put some constraints on the types of names that are accepted for registration. The protection on context objects enforced by the database system can serve to restrict the registration of undesirable names in many cases. In addition, constraints may be placed on the structure of a name's remainder. Typically, the remainder should be a simple label of the name, and not a structured component. For instance, if the name in the previous example were "A.B.C.D" then adding an authority tuple for "C.D" may be undesirable; perhaps the name should be only accepted if a context for "C" already exists.

Several options exist for choosing the set of authoritative name servers for newly registered objects. A parameter could be added to the **Register** routine to allow clients to explicitly specify a desired set of authorities. However, clients probably are not interested in such levels of detail, while system administrators are interested in keeping balanced loads on the various servers. A set of default servers could be assigned as a simple scheme. A better method would be to search for an arbitrary "Authorities" tuple in the current context and assign the same authorities to the new object. In this way, objects within the same context would tend to have identical authorities, often the authorities for the context itself.

After registration, the named object has been assigned authoritative name servers, though it has no other attributes. Only the authoritative servers for the updated context are affected by the registration. The time required for the new object to be observed by the complete name space depends on the algorithms employed for updating replicated database objects and the degree of consistency provided.

Registering a new context requires adding an "Authorities" attribute for it to the name service configuration database. Also, in order for the context to be useable, one or more "ContextBinding" attributes must refer to it. Initially, the context will be empty, though, once it is registered, object names may be inserted into it. Facilities for deleting a context are straight-forward provided that all names belonging to the context have been previously deleted.

4.3.3 Name service reconfiguration

As the distributed computing community grows over time, it will occasionally be necessary to reconfigure the name service to balance the demands placed upon it or to add new servers to offload existing servers that have become overloaded. Since the assignment of object names is independent of the assignment of responsibility for maintaining information about the objects, the name service can be easily reconfigured. That is, new name servers can be added to the environment and assume authority over part of the existing name space; application programs which rely on the name service are unaffected since the object names do not reflect the name server configuration.

Changing the authorities for a named object is more than just changing the "Authorities" attribute for that object. New authorities must acquire the complete set of attribute tuples for the object by establishing a connection to an authoritative server and retrieving the attributes. Problems may result if the "Authorities" tuple is updated before the transfer actually takes place unless the servers are prepared to try a different authority if the first does not have the desired data. To be safe, the set of authorities should be updated first in the case of a delete; the server that is no longer authoritative can then delete the object's attributes at its leisure. When adding new authorities, the authorities list should be updated after the attributes have been transferred.

Lastly, putting a new name server into service requires introducing the new server to all existing servers in the worst case, a potentially expensive operation. As an optimization, the new server's internet address need only be known by servers that contain context objects which reference contexts or objects over which the new server has authority. For a strictly hierarchical name space, this means that only servers who have direct authority over the new server need be informed of its existence. Thus, the update activity can once again be limited to a small area for well defined name spaces.

4.4 Summary

Name server configuration data enables the name resolution activity to migrate around the environment from server to server until a name is completely resolved. The configuration database, consisting of authority data and context bindings, is itself distributed and replicated so that the size of the overall name space does not place undue requirements on any single name server. The process of resolving names is inherently independent of the structure of names, although the name service administrator, when configuring the name space, may choose to exploit the structure of names to reduce the size of the configuration database.

Specifically, the following concepts play an important role in structure-free name resolution:

Authority attributes enable an object's attributes to be located.

Context objects allow the set of authority attributes to be partitioned and distributed.

Clustering conditions serve as criteria for assigning authority attributes to context objects.

Context bindings allow names to be resolved.

The *policy* for resolving names, as represented in the configuration database, is separated from the *mechanism* for resolving names. Three styles of name resolution, recursive, iterative, and transitive, place different computation and communication requirements on the name servers and name agents.

The mechanisms supporting this new approach to name management are more complicated, and hence more expensive, than existing schemes for resolving names based solely on their structure. However, the added flexibility allows name spaces for large computing environments to evolve over time. Since the configuration data is stored as attributes of objects, just like any other name server data, the name service can be easily configured and reconfigured. Space/time tradeoffs exist in which the amount of storage dedicated to configuration data can be reduced if the resolution chain is lengthened for some names. On the other hand, the name resolution chains can be reduced compared to existing name resolution schemes by dedicating more storage to configuration data.

Chapter 5

Performance Analysis

An analytical model for distributed name services allows one to investigate the effect of various design and configuration choices on the cost of name service operations. Although a name service plays a vital role in internetwork environments, few attempts have been previously made in the computing literature to quantify the performance of distributed name resolution. New results show that the cost of name service operations with a decentralized service need not be appreciably greater than with a centralized service (though more storage space is required for configuration data). Applying the simple performance model to a sample environment indicates that substantial cost benefits can be accrued through replication of name service data; however, the benefits depend heavily on the topology of the environment. For a moderate degree of replication, the unavailability of a few name servers does not significantly increase the costs of name service operations, ignoring increased server congestion.

5.1 Name Service Performance

The cost of communication between clients and name servers is the major bottleneck in locating remote resources in environments consisting of a ·ubstantial number of interconnected networks with a large number of hosts. In such an environment, the performance of name server operations is dominated by the number of name servers that must be accessed and the cost of accessing those name servers. The name service, that is, the group of name servers that collectively manage the name space, should be configured so as to minimize the cost of name service operations for the average client.

Once a naming convention has been adopted, the many factors affecting the efficiency with which the name space can be managed and the cost of performing operations on name server information include:

- the performance of each individual name server,
- the placement of name servers throughout the internet,
- the amount of replication of name server information,
- the choice of authoritative name servers for parts of the name space,
- the number of name servers that are currently operational,
- the clients' patterns of reference to name server information.

These factors, with the exception of the last one, characterize the current configuration of the name service. This chapter looks at each of these issues in detail.

The next section presents a simple model of distributed name services that enables the cost of a name server operation to be quantified for a variety of name server configurations and name resolution policies. The model does not attempt to give detailed performance predictions, such as those that might be obtained through simulations, but rather concentrates on analytical formulations of the high-level interactions required between name servers to complete an operation. The goal is to be able to compare the cost of operations for different choices that must be made by administrators when configuring a name service.

In practice, the cost formulas derived for name server operations can be applied to existing environments to analyze and subsequently improve the performance of the system, or they can aid in making design decisions when configuring a new system. For instance, a network administrator may wish to assess the benefits of increased replication or the addition of a new name server.

5.2 A Model for Name Server Interaction

5.2.1 Name servers and clients

A name service consists of N servers, $NS_1 \dots NS_N$, distributed throughout an internet. At any point in time, some fraction of these servers will be accessible; the others may have crashed or become detached from the network. S_F represents the current set of name servers whose data is inaccessible because of some failure; $S_F \subset \{NS_1, \dots, NS_N\}$ has cardinality F.

The various name server clients are enumerated $1 \dots U$. The term "client" may refer to a specific program, host, network, or some combination thereof. In general, clients are distinguished by their location in the internet relative to the name servers and by the particular objects they reference.

A name service client need only know the location of a single name server, presumably the closest one, to make use of the name service. Name service operations are assumed to be performed iteratively: if the primary name server, NS_{main}, is unable to resolve a name, then it returns the location of a more knowledgeable colleague. Several iterations may be necessary to perform an operation for some naming conventions and management strategies.

5.2.2 The network

The round trip transmission cost between client u and name server i is given by c_{ui}. Observe that c_{ui} strongly depends on the sites at which the client and server are executing. It varies according to the number of gateways traversed and the speeds of the intermediate transmission lines. The number of bytes transferred is assumed to have a negligible influence on the communication cost since name server queries and responses are generally quite small.

This is a very simple static measure of the cost of communication between clients and servers. In particular, variations due to network congestion are ignored. While such a model may be reasonable for widely distributed environments with slow speed lines and many gateways, it certainly would not suffice for local area networks. The model does not include the cost of communication between servers since an iterative style of name resolution is assumed.

5.2.3 The database

The name service database is strictly partitioned into K database objects. In the degenerate case, each database entry is a separate object. The database objects, $db_1 \dots db_K$, correspond to indivisible units of storage. That is, either the complete database object is stored at a given name server or none of it is.

Each name server has authority over some subset of the database partitions. Typically, no single name server stores the complete database. The set S_k contains those name servers that store object db_k; S_k, for $k = 1 \ldots K$, is a subset of $\{NS_1, \ldots, NS_N\}$. In other words, S_k is the list of authoritative name servers for db_k.

For each name server, d_i denotes the cost of executing a database operation at NS_i. For simplicity, this cost, which could depend on such things as the overall size of the database maintained by NS_i and the kind of database facilities employed, is assumed to be fixed over time. In particular it does not account for variations in the load at the server. Moreover, no distinction is made between different types of database operations.

5.2.4 Reference patterns

Each client has a set of objects (or resources) that it regularly references. Different clients generally perform different name service operations on sets of objects with varying frequencies. Client u's reference mix is represented by $r_{u1} \ldots r_{uK}$. That is, r_{uk} is the percentage of name server accesses performed by client u to the database partition db_k. Note that the r_{uk}'s characterize a client's *logical access patterns*.

Physical access patterns, the fraction of accesses to individual name servers by each client, are dependent on both the frequency of accesses to the name service database entries (r_{uk} for $k = 1 \ldots K$) and the mapping of data to storage sites (S_k for $k = 1 \ldots K$). The *locality of reference* is the degree to which local name servers are accessed more frequently than distant servers. Locality in the physical access patterns is desirable since local servers can be accessed more cheaply than distant name servers. The amount of locality achievable in practice depends on the distribution of clients that are interested in a particular name server entry.

5.2.5 Operation costs

For a given name server, NS_i, C_{ui} specifies the cost of accessing that name server remotely from client u. This cost includes both the communication and processing costs. Hence, C_{ui} is the sum of d_i and c_{ui}.

For a particular operation $o \in \{lookup, update\}$, L_{ouk} represents the total cost of performing operation o by client u on information in database object db_k. For a centralized name service with a single server, NS_{main}, L_{ouk} would be simply $C_{u\,main}$. However, for a distributed service, L_{ouk} includes the cost of locating the desired data; this name resolution cost may involve retrieving configuration data from one or more name servers. L_{ouk} is often denoted as simply L_{uk} in cases where the particular operation is clear from context or where $L_{lookup\;uk} = L_{update\;uk}$.

The complete cost of operating on name server information, such as performing a name lookup, varies per client according to the client's location relative to the various name servers and the client's reference mix. The expected value of this cost for client u, denoted by $E(L_u)$, is weighted according to the client's reference mix:

$$E(L_u) = \sum_{k=1}^{K} r_{uk} E(L_{uk}). \tag{5.1}$$

Deriving an optimal configuration would involve minimizing the sum of the expected costs for all clients.

5.2.6 Summary

This section advanced a model for distributed name services. The parameters of the model, which characterize the name service's configuration, are summarized in Figure 5.1. When applying this model to study a proposed name service configuration, system administrators have control over

$NS_1 \ldots NS_N \overset{\mathrm{def}}{=}$ set of name servers

$1 \ldots U \overset{\mathrm{def}}{=}$ name server clients

$db_1 \ldots db_K \overset{\mathrm{def}}{=}$ name server database objects

$S_k \overset{\mathrm{def}}{=}$ set of authoritative name servers for db_k

$r_{uk} \overset{\mathrm{def}}{=}$ fraction of client u's accesses to db_k

$c_{ui} \overset{\mathrm{def}}{=}$ cost of communicating with NS_i from client u

$d_i \overset{\mathrm{def}}{=}$ cost of performing an operation at NS_i

$C_{ui} \overset{\mathrm{def}}{=} c_{ui} + d_i$

$S_F \overset{\mathrm{def}}{=}$ set of failed name servers

Figure 5.1: **Name service model parameters.**

N, db_k, and S_k. The parameters, d_i, r_{uk}, and c_{ui}, to a large extent, should be measured or projected. The communication costs, c_{ui}, however, also depends upon the placement of servers, which can be controlled.

Clients and system designers are primarily interested in the expected name service operation cost $E(L_u)$, which is a function of these parameters. Studying the effects of various configuration choices can be accomplished by varying a parameter, while holding the others constant, and observing changes in the expected cost. Typically, the cost values for the parameters are specified in units of time so that L_u gives an expected level of performance. Alternative measures of cost, such as dollars, could also be used.

5.3 Performance of Individual Servers

The model is not concerned with being able to predict the performance of a particular name server since standard performance evaluation and improvement techniques can be applied to analyze and enhance an individual server's level of performance. Also, additional name servers can be employed if existing ones become overloaded. Instead, the performance of various servers is used indirectly to gauge the distributed performance of the overall service. In the model, NS_i's performance is completely embodied in the database operation cost, d_i, and the processing component of the communication costs between clients and the server, c_{ui}.

5.4 Name Server Placement

Generally, the placement of name servers in the distributed environment is dictated by administrative considerations, rather than by performance. An organization provides the name servers required to manage the objects created and owned by members of that organization, or else arranges to lease time and storage from another organization's server. The location of servers has an indirect influence on performance through the database objects that are assigned to particular name servers and the cost of communicating with these servers. This influence may be substantial for very large distributed communities.

As an example of a widely distributed environment, consider the network topology of the Grapevine system (as of summer 1983 [Schroeder *et al.* 84]) presented in Figure 5.2. The circles represent Ethernet local area networks, while the lines are long distance links with data rates of

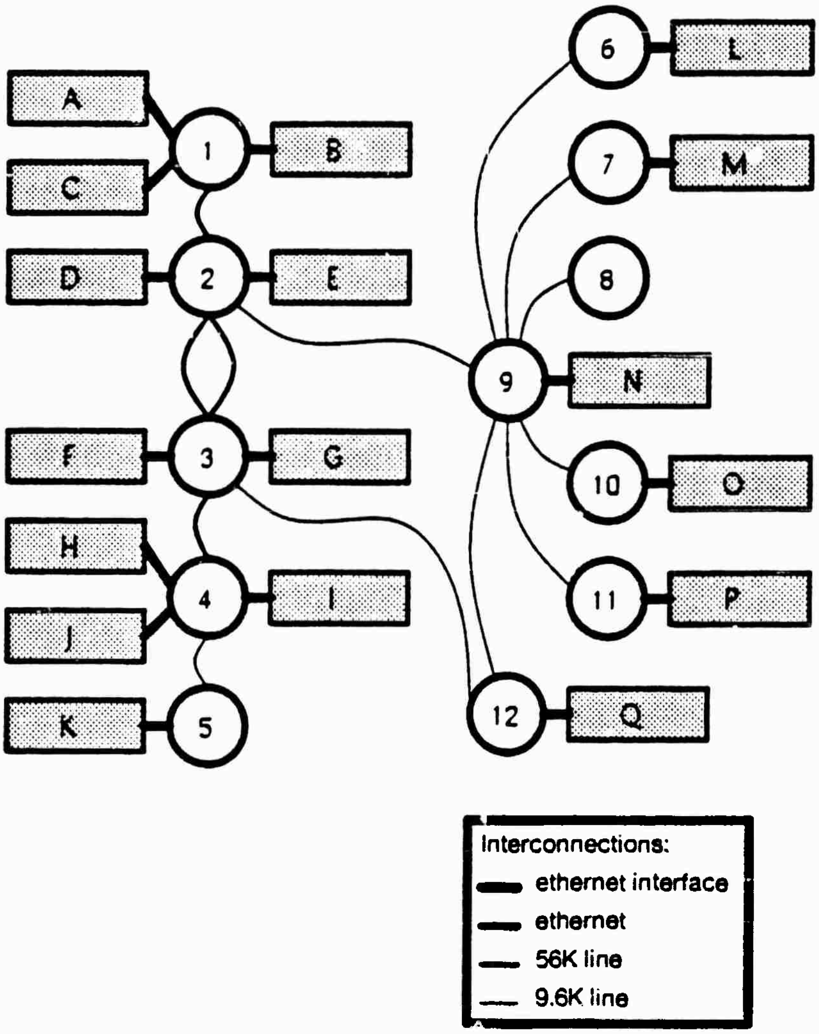


Figure 5.2: A sample internet.

This is the configuration of Grapevine servers in everyday use at the Xerox Palo Alto Research Center as of summer 1983.

from	to server								
network	A	B	C	D	E	F	G	H	I
1	1	1	1	56	56	111	111	166	166
2	56	56	56	1	1	56	56	111	111
3	111	111	111	56	56	1	1	56	56
4	166	166	166	111	111	56	56	1	1
5	479	479	479	424	424	369	369	314	314
6	682	682	682	627	627	682	682	737	737
7	682	682	682	627	627	682	682	737	737
8	682	682	682	627	627	682	682	737	737
9	369	369	369	314	314	369	369	424	424
10	682	682	682	627	627	682	682	737	737
11	682	682	682	627	627	682	682	737	737
12	424	424	424	369	369	314	314	369	369

from	to server							
network	J	K	L	M	N	O	P	Q
1	166	479	682	682	369	682	682	424
2	111	424	627	627	314	627	627	369
3	56	369	682	682	369	682	682	314
4	1	314	737	737	424	737	737	369
5	314	1	1050	1050	737	1050	1050	682
6	737	1050	1	627	314	627	627	627
7	737	1050	627	1	314	627	627	627
8	737	1050	627	627	314	627	627	627
9	424	737	314	314	1	314	314	314
10	737	1050	627	627	314	1	627	627
11	737	1050	627	627	314	627	1	627
12	369	682	627	627	314	627	627	1

Table 5.1: **Communication costs.**

Entries are derived for the internet depicted in Figure 5.2 and listed in units of T, where T represents the cost of communicating over a local ethernet.

either 56 kilobits/second or 9.6 kilobits/second. The local networks are numbered from 1 to 12. The rectangles depict the various name servers, labeled from A to Q.

In an existing environment of this sort, the values for c_i could be easily obtained from measurement studies. For the sake of example, suppose that estimates for these quantities are needed, as would be required if the system were in the planning stages. Table 5.1 enumerates the costs of communicating between a client on each network and each name server using the following simple algorithm: Communication costs are normalized so that communicating over a local Ethernet incurs one unit of cost, denoted by T. Assuming that the communication cost is proportional to the data transmission rate of the communication medium[1], transmission over a 56K bps line costs approximately $54T$, and similarly, communication over a 9.6K bps line costs around $312T$. The host to host communication costs, then, are derived by adding the costs of the various communication links traversed; added costs due to delays in the gateways have been ignored.

Notice that the costs of communicating between a client and various servers may differ by several

[1] This assumption is made solely for the sake of example. Studies show that the cost of communication over high-speed local networks actually bears little relationship to the bandwidth. For long-haul slow-speed communication lines, however, the assumption used in this example is more realistic.

orders of magnitude. Fortunately, the distribution of database objects among servers can alleviate much of these differences by storing name server information close to where it is frequently used.

5.5 Assigning Authority

5.5.1 Basics

The association between an adopted naming convention and the assignment of authority for managing the name space has been previously explored [Terry 84]. This section uses the model of a distributed name service to quantify the cost implications of various classes of existing naming conventions. The analysis assumes that a single copy of each database object exists; the benefits of replicating database objects are studied in a later section. Since the cost of lookups and updates are identical under this assumption, the analysis is worded in terms of name server lookups without loss of generality.

Although a client's reference mix, which the system designer has no control over, contributes significantly to the client's expected name server lookup cost, it plays no part in the cost of retrieving or updating an individual object's attribute. Thus, the following analysis concentrates on formulating L_{uk}, and ignores the clients' access patterns in $E(L_u)$. The client subscript u is left out of the formulas to increase their clarity; this can be safely done since the performance observed by a particular client is independent of the locations of other clients. It should be kept in mind that L_k, which really varies from client to client, is a shorthand for L_{uk}, and C_k is a shorthand for C_{uk}.

5.5.2 Flat name space

To start with a simple case, consider managing a flat name space. The two basic alternatives are giving a single name server authority over the complete name server database or choosing an arbitrary authority for each database object and using broadcast or searches to resolve names. In Chapter 2, both of these approaches were ruled out for performance reasons, among others.

In the first case, with $S_k = \{NS_{central}\}$ for all k, the name server can perform any operation since it contains the complete set of information about all named objects in the environment. Thus, the retrieval cost is simply

$$L_k = C_{central}. \tag{5.2}$$

This approach appears very attractive in the cost of name server lookups, though, in reality, the single name server would have to be centrally located in the environment and hence the cost for accessing it, $C_{central}$, would generally be much greater than the cost of accessing the closest server in a distributed name service, C_{main}.

For $S_k = \{NS_i\}$ with the authoritative name server for an object chosen at random, if the name service contains no configuration data, locating the desired attribute may necessitate querying each name server in succession until the authoritative one is discovered. The retrieval cost becomes

$$L_k = \sum_{j=1}^{i} C_j, \tag{5.3}$$

assuming that the name servers are queried in numerical order. This second approach is costly in terms of name server interactions since half of the name servers must be accessed on the average to retrieve the object's information. As noted earlier, neither approach is very practical for large environments.

If authority attributes are introduced, so that the set S_k is maintained at all name servers for all database objects while the database itself remains distributed as proposed in section 3.2.2, then the cosf can be reduced to

$$L_k = C_{main} + d_{main} + C_i. \tag{5.4}$$

The first interaction (C_{main}) resolves the name while the second (C_i) performs the desired operation; the extra database access (d_{main}) is required to retrieve the internet address of the authority.

At first glance it may appear that the cost in Equation 5.4 has more than doubled that of Equation 5.2 for a centralized server. Actually, $E(C_i)$ is approximately the same as $C_{central}$ assuming that data is distributed randomly and referenced with equal likelihood. Moreover, $C_{main} = min\{C_1, \ldots, C_N\}$, so the difference may be quite small. With locality of reference, studied in section 5.8.1, $E(C_i)$ could be significantly less than $C_{central}$, and hence, the lookup cost for a distributed name service may actually be less than that for a centralized service.

5.5.3 Physically partitioned name space

With a physically partitioned name space, a one-to-one mapping exists between database objects and name servers. That is, $K = N$ and $S_k = \{NS_k\}$. Even though the authority for an object can be explicitly recognized from its name, two accesses are required to perform a name server operation: one to locate the naming authority and one to access the data. A special case arises if the desired naming information is stored at the primary name server; in this case, only a single access is required since the main name server can recognize that it is the authority and return the data directly. The cost of a lookup is thus

$$L_k = \begin{cases} C_{main} + C_k & \text{if } k \neq main, \\ C_{main} & \text{if } k = main. \end{cases} \tag{5.5}$$

However, if the total number of name servers is small, clients can easily cache the network addresses of the various name servers, thereby reducing the cost to

$$L_k = C_k. \tag{5.6}$$

The access to the local name server has been eliminated since the individual hosts are knowledgeable enough to query the correct storage site directly. The resulting lookup algorithm is optimal given the assumption that naming data is stored exactly once.

5.5.4 Organizationally partitioned name space

Suppose the name space is partitioned according to administrative organizations and that each organization's data is managed by a single name server, $S_k = \{NS_i\}$. If each name server knows which server has responsibility for each organization, name server queries can be processed in two steps as before. First, a client's local name server maps the organization name to the authoritative name server for that organization and returns its network address. Then the remote name server is contacted to retrieve the appropriate naming information. The lookup cost is basically the same as Equation 5.4 for a flat name space using authority attributes for name resolution,

$$L_k = C_{main} + d_{main} + C_i. \tag{5.7}$$

The major difference is that the organizational clustering serves to reduce the total amount of configuration data compared to a flat name space with an authority attribute per name.

Note that, unlike physically partitioned data, two database retrievals are always required since a name server can not determine whether or not it is the authority for the desired data without consulting the local database. One round trip transmission cost can be saved, however, if the primary name server retrieves and returns the name server entry directly upon discovering that it is the storage site for the desired data. Thus,

$$L_k = C_{main} + d_{main} \qquad \text{if } S_k = \{NS_{main}\}. \tag{5.8}$$

Often, rather than all of an organization's objects having identical authorities, the authority for objects is distributed within an organization. In this case, the authority for each organization contains information about which servers are authoritative for objects within the organization; the initial context need only contain a list of authorities for top-level organizations. The resolution chain for names is thus increased in length, and the cost of an operation becomes,

$$L_k = ((C_{main} + d_{main}) + C_{org} + d_{org}) + C_i. \tag{5.9}$$

The nesting in the formula corresponds to the iterative name resolution calls.

The analysis for longer name resolution chains is a straightforward extension. If the name resolution activity performs a context binding at the list of servers, $NS_{i_1}, NS_{i_2}, \ldots, NS_{i_t}$, where $i_1 = main$, then,

$$L_k = \sum_{j=1}^{t}(C_{i_j} + d_{i_j}) + C_i. \tag{5.10}$$

Therefore, assuming database objects are uniformly distributed throughout the environment, the expected cost of retrieving an attribute stored in database object db_k is given by,

$$E(L_k) = ((C_{main} + d_{main}) + (t-1)(\overline{C + d})) + C_i. \tag{5.11}$$

Of course, each step in the resolution chain does not necessarily require communication between the client and a name server. For example, if $NS_{i_j} = NS_{i_{j+1}}$ for some $1 \leq j < t$, then the communication cost c_{i_j} can be avoided. Thus, the formula given for L_k in Equation 5.10 can be considered an upper bound on the cost of a name server operation.

5.6 Benefits of Replication

Assuming that a read-any/write-all algorithm for replicated data is adopted, replicating database objects decreases the cost of name server lookups, but increases the cost of update operations. The main cost of increased replication results from the need to maintain consistency among the various copies of a database object when updates are applied to the object. Although the update cost depends on the exact algorithm employed for maintaining consistent replicated copies, a simple estimate can be obtained by adding the costs of performing an update at each individual authoritative name server. For an organizationally clustered name space, the update cost can then be estimated by,

$$L_{update\,k} = C_{main} + d_{main} + \sum_{NS_i \in S_k} C_i. \tag{5.12}$$

Observe that the update cost is an increasing function of the degree of replication.

On the other hand, with replicated data, any available copy of an organization's name server data can be used to answer queries. For performance reasons, accessing the closest authoritative name server for the named object is generally desirable. Assuming that the closest authoritative name server, $NS_{min_k} \in S_k$, can be determined with negligible cost[2], the name server lookup cost becomes

$$L_{lookup\,k} = C_{main} + d_{main} + C_{min_k}. \tag{5.13}$$

This formula looks similar to previous formulas, such as Equations 5.4 and 5.7. However, the cost should be less with replicated data since the name server accessed by various clients, NS_{min_k}, could

[2] For a large environment with a substantial number of name servers, determining the closest server may not always be feasible. In the Grapevine system, each registration server maintains a complete list of the other servers ordered by distance [Schroeder *et al.* 84]. For other environments, it may be sufficient for a server to keep lists of neighboring servers: if none of the neighbors are authoritative for the current context or object, meaning all authorities are distant, then an authority could be arbitrarily chosen without unduly impacting performance.

client's	replication factor R =					
network	1	2	3	4	5	6
1	297.35	145.99	84.83	56.72	41.53	32.15
2	261.76	124.96	74.40	53.69	43.39	36.94
3	271.47	125.76	74.24	53.63	43.38	36.94
4	300.59	142.34	82.65	55.96	41.34	32.12
5	576.76	398.65	323.48	281.71	251.52	226.25
6	645.18	548.70	488.14	435.57	387.66	343.13
7	645.18	548.70	488.14	435.57	387.66	343.13
8	976.59	896.22	849.47	808.41	769.71	732.08
9	369.00	307.04	278.71	256.05	235.77	216.55
10	645.18	548.70	488.14	435.57	387.66	343.13
11	645.18	548.70	488.14	435.57	387.66	343.13
12	439.41	347.85	302.26	271.68	246.62	223.93
avg.	506.14	390.30	335.21	298.34	268.66	242.46
Δ%	—	-22.89	-14.11	-11.00	-9.95	-9.75

Table 5.2: **Effects of replication on lookup costs.**

Entries give the expected cost of a name service lookup in units of T.

differ from client to client, whereas before each client was forced to access the same server. Nevertheless, without some knowledge of how the authoritative name servers are selected and how many exist for a given database partition, comparing the costs of the different name space management techniques is very difficult.

One simple approach would be to distribute R copies ($R \leq N$) of each database object uniformly. In other words, R authoritative name servers are chosen at random for each named object. Without loss of generality, assume for the moment that the name servers are ordered relative to a particular client such that $C_i \leq C_j$ for $i < j$ and $NS_1 = NS_{main}$. Under this assumption, the expected lookup cost can be computed as follows,

$$E(L_{lookup\,k}) = C_{main} + d_{main} + \sum_{i=1}^{N} Prob(i = min_k)C_i \tag{5.14}$$

$$= C_{main} + d_{main} + \sum_{i=1}^{N} \frac{\binom{N-i}{R-1}}{\binom{N}{R}} C_i. \tag{5.15}$$

This formula allows one to quantitatively determine the benefit of replication on performance by increasing the value of R. Of course, the benefits that can be achieved depend greatly on the physical configuration of the internet and the placement of the name servers.

The random selection of a fixed number of storage sites for database objects is a particularly naive configuration technique. Generally, if the client reference patterns are known, the cost of lookup operations can be reduced by distributing data intelligently to coincide with its regions of interest. The cost formula derived above in Equation 5.15 based on randomly selected storage sites thus provides a good indication of the minimum achievable performance.

Using the configuration in Figure 5.2 and the associated estimates of communication costs given in Table 5.1 along with Equation 5.15, Table 5.2 presents the effects of replication on the expected performance of name server retrievals assuming that copies are uniformly distributed. In this example, the database access cost, d_i, is taken to be $6T$ in accordance with experience indicating that, for retrieval over a local network, the cost of database queries generally dominates the communication cost by about a factor of 4 to a factor of 8. The number of copies of each partition has been varied from 1 to 6. The expected cost of a name server query is computed for a client on each of the 12 networks and then averaged over all networks. The last line of the table indicates the change in the average expected cost resulting from an additional copy of each database partition.

On the average, having two copies of the data instead of one reduced the expected lookup cost by over 22%. For networks 1-4, which a e connected by high speed lines, improvements of over 50% are achieved. Notice that clients on network 8, which has no local name server and is separated from the rest of the world by low speed lines, suffer the worst performance. Furthermore, replication does not help them as much as others. Networks 1 and 4, which have three local name servers apiece, benefit the most from replication. In all cases, adding an extra copy of the name server data has a substantial impact on performance regardless of the replication factor. These performance increases are due entirely to reducing the amount of communication between clients and very remote name servers.

5.7 Name Server Failures

With partially redundant name server data, the failure of a name server should potentially degrade performance, but should not render any information unavailable provided the number of failures is less than the degree of replication. If the number of name server failures, F, exceeds the degree of replication, R, then all responsible name servers for the information may have crashed. The probability that a given piece of data is inaccessible becomes

$$Prob(data\ inaccessible) = \prod_{l=0}^{R-1} \frac{F-l}{N-l}$$

which is always zero for $F < R$.

Basically, failures introduce a variability in the degree of replication of database objects. Not only do different database objects have different numbers of available copies depending on which servers are down, but also a given object's degree of replication varies over time.

The effect of name server failures on performance can be gauged by incorporating such failures into the previous lookup cost formula. For the set of failed name servers selected at random, S_F with $F < R$, the lookup cost formula remains as in Equation 5.14,

$$E(L_{lookup\,k}) = C_{main} + d_{main} + \sum_{i=1}^{N} Prob(i = min_k)\ C_i, \tag{5.16}$$

But the probability of retrieving the desired resource information from name server i, $Prob(i = min_k)$, becomes substantially more complex. The name server operation on db_k is performed at NS_i if and only if name server i stores the data ($NS_i \in S_k$), is still alive ($NS_i \notin S_F$), and all authoritative servers that are closer to the requesting client are inaccessible ($NS_j \in S_F\ \forall j$ *such that* $NS_j \in S_k \cap j < i$).

The probability of name server i being available is simply,

$$1 - \frac{F}{N}.$$

Combinatorics says the probability that q authoritative name servers are closer than NS_i is given by,

$$\frac{\binom{i-1}{q}\binom{N-i}{R-q-1}}{\binom{N}{R}},$$

client's network	number of failures F = 0	1	2	3	4
1	41.53	46.00	51.42	58.08	66.36
2	43.39	46.42	50.22	55.06	61.32
3	43.38	46.39	50.17	55.01	61.31
4	41.34	45.64	50.83	57.21	65.19
5	251.52	260.40	270.13	281.03	293.53
6	387.66	401.75	416.19	431.01	446.33
7	387.66	401.75	416.19	431.01	446.33
8	769.71	781.09	792.65	804.42	816.53
9	235.77	241.73	247.87	254.24	260.92
10	387.66	401.75	416.19	431.01	446.33
11	387.66	401.75	416.19	431.01	446.33
12	246.62	253.99	261.77	270.09	279.14
avg.	268.66	277.39	286.65	296.60	307.47
Δ%	—	3.25	3.34	3.47	3.66

Table 5.3: **Effects of failures on lookup costs for** $R = 5$.

Entries give the expected cost of a name service lookup in units of T.

while the chances that all q of them are dead given that NS_i is alive is,

$$\prod_{j=0}^{q-1} \frac{F-j}{N-1-j}.$$

Putting this all together and enumerating over possible values of q,

$$Prob(i = min_k) = \sum_{q=0}^{R-1} \frac{\binom{i-1}{q}\binom{N-i}{R-q-1}}{\binom{N}{R}} [1 - \frac{F}{N}] \prod_{j=0}^{q-1} \frac{F-j}{N-1-j}. \tag{5.17}$$

Returning to the sample distributed computing environment in Figure 5.2, Table 5.3 presents the effect of failures on the cost of retrieving name server information. Again, the results are given for clients on each network and averaged over all networks. These results indicate that name server failures actually degrade performance by very little for a replication factor of 5. Even if almost one fourth of the name servers are down, the expected lookup cost increases by only 15% on the average, and around 50% for the worst case.

The availability of name server data, not performance, appears to be the primary concern when considering name server failures. However, recall that the simple name server model used in this chapter assumes that the load on servers does not vary over time. With failures, added congestion at servers would likely increase the cost of name service operations more than the analytical results suggest.

5.8 Exploiting Client Behavior

5.8.1 Locality of reference

A name service client's behavior is characterized by the frequency of operations it performs and the database objects those operations affect. Recall that a particular client's reference mix is represented by a list of the probabilities of accessing individual database objects, $r_{u1} \ldots r_{uK}$ for client u, and that the effect of the client's referencing behavior on its expected operation cost is as given in Equation 5.1:

$$E(L_u) = \sum_{k=1}^{K} r_{uk} E(L_{uk}). \tag{5.18}$$

Locality of reference occurs if the most frequently accessed database objects are those that can be operated on with the lowest expected cost, generally objects that are in the proximity of the client.

If the client references all database objects with equal likelihood, $r_{uk} = r_{ul}$ for all $1 \geq k, l \leq K$, then the overall expected operation cost does not depend on the particular assignment of authority, assuming that all name servers are assigned the same number of database objects. Even if database objects are referenced with varying frequencies, the expected lookup cost remains independent of the client's particular reference mix as long as the assignment of authority for database objects is performed arbitrarily.

As an example, for a simple organizationally partitioned name space, the expected cost obtained from plugging Equation 5.7 into Equation 5.18 is

$$E(L_u) = \sum_{k=1}^{K} r_{uk} E(C_{u\ main} + d_{u\ main} + C_{ui}). \tag{5.19}$$

$$= \sum_{k=1}^{K} r_{uk} (C_{u\ main} + d_{u\ main} + E(C_{ui})).$$

If the authority for database objects is randomly distributed among name servers, that is, the storage site for a database object is chosen arbitrarily, then $E(C_{ui}) = \overline{C_u}$ and Equation 5.19 becomes

$$E(L_u) = (C_{u\ main} + d_{u\ main} + \overline{C_u}) \sum_{k=1}^{K} r_{uk} \tag{5.20}$$

$$= C_{u\ main} + d_{u\ main} + \overline{C_u}.$$

Note that this expected cost is independent of the values for $r_{u1} \ldots r_{uK}$.

Substantial gains in the expected operation cost can only be achieved by storing data close to where it is frequently used. In other words, $E(L_u)$ is reduced if for two database objects db_k and db_l, $L_{uk} < L_{ul}$ when $r_{uk} > r_{ul}$. Fortunately, localities of interest naturally arise in large distributed systems. For example, clients residing in a local environment, such as Berkeley, are presumably most often interested in objects created within that environment, and much less frequently interested in referring to distant objects. The assignment of authority for storing database objects should be done intelligently to exploit the measured or expected locality of interests. Replication can be used in cases where two geographically-distant clients share certain localities of interest. Chapter 6 discusses the results of an experiment to measure the locality present in the Grapevine system.

5.8.2 Lookup/update ratio

A second aspect of clients' referencing behavior that can be exploited to reduce the expected name service costs is the frequency of various operations, such as the ratio of update to lookup operations. While $E(L_u)$ is the expected cost of performing a given name service operation, the overall cost incurred by a particular client, $E(TOTAL_u)$, is the sum over all operations, weighted by the probabilities of those operations. For the two operations, lookup and update, this is given by

$$E(TOTAL_u) = Prob(lookup)E(L_{lookup\ u}) + Prob(update)E(L_{update\ u}) \tag{5.21}$$

where $Prob(lookup) + Prob(update) = 1$.

Generally, techniques that reduce the expected cost of one operation increase the cost of another. This is, choices can be made that trade off the costs of different operations. For example, as demonstrated earlier in this chapter, increasing the replication factor of database objects improves the cost of lookups, but renders updates more expensive. The proper choices for configuring the name service thus depend on the expected ratio of operations, $Prob(lookup)/Prob(update)$. For $Prob(lookup) >> Prob(update)$, efforts should be made to reduce $E(L_{lookup\ u})$, and vice versa for $Prob(lookup) << Prob(update)$.

Given that name services are primarily used to locate and maintain information about named objects, and that long-lived objects move infrequently, one would expect name service lookups to be much more prevalent than updates. Therefore, one's intuition would be to optimize the cost of name service lookup operations at the expense of updates.

5.9 Summary

Once a naming convention and associated name space management strategy have been selected, the observed performance of name service operations is dictated primarily by the placement of the name servers, the distribution of the name service database, and the patterns of reference to name service information. The simple analytical model of a distributed name service presented in this chapter allows one to quantitatively measure the high-level impact of a name service's configuration on a given client's level of performance.

Since the costs of communicating with name servers in a large distributed computing environment may vary from client to client and server to server by several orders of magnitude, minimizing the number of interactions with servers and localizing those interactions is the key to low operation costs. The name management policy adopted determines the amount of communication required to resolve a name and access the appropriate database object. Reducing the cost of this communication is achieved mainly through replicating name service data and exploiting inherent localities of client references.

The random selection of a fixed number of storage sites for database objects was analyzed as a particularly naive configuration technique. For such a scheme, the degree of replication of database entries was shown to considerably impact the cost of accessing a given database entry. In cases where some locality of reference exists, and data is distributed intelligently so as to coincide with its regions of interest, the cost formulas based on randomly selected storage sites can serve as a lower bound on performance.

Chapter 6

Measurements of Grapevine

Experimental measurements of Xerox's Grapevine registration service indicate properties of clients' reference patterns that can be exploited to enhance performance, including large localities of interest. The ratio of name service lookups to updates initiated by electronic mail clients, which is high for individuals, is surprisingly low for group names in Grapevine. The measurements, used as inputs to the model presented in the previous chapter, demonstrate the benefits of intelligent name service configuration and client reference locality on name service response times.

6.1 Basics of the Experiment

6.1.1 Goals

The previous chapter discussed several aspects of clients' behavior that have drastic influences on the performance of name service operations. It also suggested ways in which, given knowledge of the clients' behavior, such behavior could be exploited to improve performance. Prompted by these analytical results, an experiment was undertaken to obtain actual measurements of the amount of reference locality that exists in a large distributed community; tabulations of the frequency of various operations performed by name service clients were also desired.

6.1.2 Why Grapevine?

The Grapevine registration service was chosen as the object of the study since it is perhaps the only widely distributed name service with a sizeable user community. Close to 5000 individuals within the Xerox Corporation use Grapevine daily to exchange electronic messages. At the time of the study, the Grapevine system consisted of 20 dedicated servers distributed throughout the continental United States, with one server in Canada and one in England. Its implementors claim that. as of the Summer of 1983, over 8,500 messages were submitted to the Grapevine mail service in a typical day [Schroeder *et al.* 84]. Figure 6.1 shows the interconnection topology of the 17 Grapevine servers that existed at this time.

Large widely distributed systems that are heavily relied upon by users to perform their daily work are difficult to modify. Adding hooks to such a system to keep statistics and obtain measurements would be painful at best. probably unacceptable. Fortunately, Grapevine servers maintain logs of their activities. Although these logs were designed as a tool to monitor and debug the system [Schroeder *et al.* 84], they contain sufficient data to derive most of the desired numbers. A snapshot of Grapevine's logs thus served as the basis for studying how Grapevine is used by its clients.

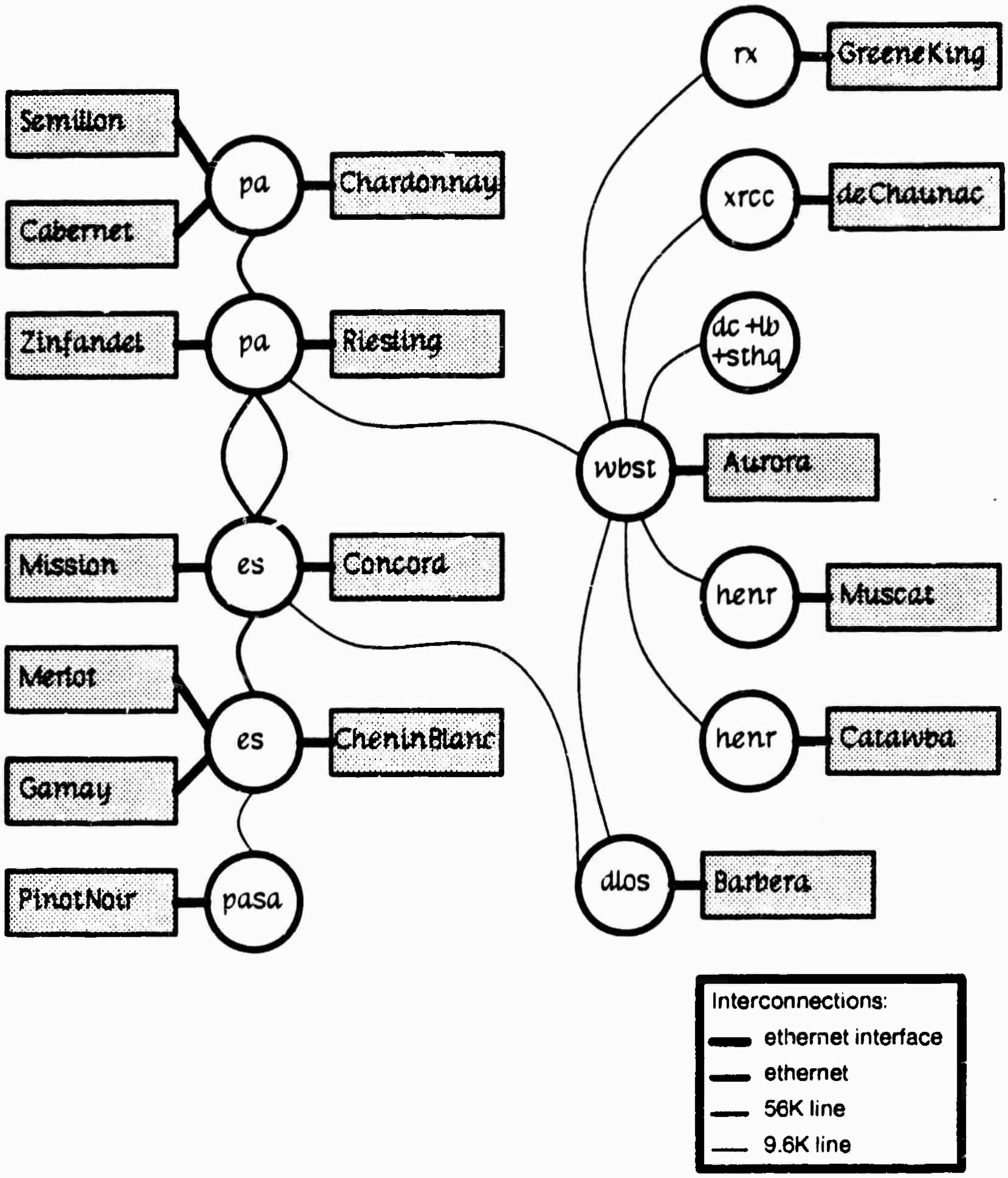


Figure 6.1: **Topology of the Grapevine internet.**

6.1.3 Grapevine's logs

Each Grapevine server keeps a local log, consisting of 120 512-byte Alto pages treated like a circular buffer [Birrell 83]. The log contains a list of one-line log records pertaining to both the registration and mail services. Date records start with octal 377 and give the current date, such as "10-Dec-83". All other log records consist of an indication of the current (local) time, relative to the last date record, followed by a description of some activity. For example, if the server "Cabernet" was booted at time 17:12:38, it would write a log record of the form,

```
17:12:38 Grapevine: Registration Server Cabernet.gv. Mail Server Cabernet.ms
```

The contents of individual log records depend on the particular activity being logged. No explicit relationship exists between adjacent log records other than their chronological ordering.

When half of the server's log fills up, the server dumps it to a file server while the other half is being used. Forty files containing full logs are kept on a file server for each Grapevine server. These files are themselves treated like a large circular buffer, that is, the dumping of a server's log causes the contents of the oldest log file to be overwritten. Forty log files (2,457,600 bytes) should be large enough to hold a week's history [Schroeder *et al.* 84].

6.1.4 Retrieving, parsing, and analyzing log data

The first phase of the study involved retrieving each server's log files from the appropriate file servers. Using the Cedar programming environment, this was accomplished from a program using a file transfer protocol; even the servers in Canada and England could be easily accessed. The logs files were then concatenated into a single file for each server, being careful to preserve the records' chronological ordering.

This provided a snapshot of Grapevine's activity for a certain period of time, the period varying from server to server based on its amount of activity. Some servers had months of log data while others had barely a week's worth. For consistency, each server's log file was pruned to span exactly one week. That is, all records outside of the range 00:00:00 PST December 4, 1983 to 23:59:59 PST December 10, 1983 were discarded[1]. This left about 20 megabytes of log data to be analyzed.

The templates for various log records can almost always be identified by the record's first word. The parser built to read the log data takes advantage of this fortuitous property in the following way: the first word of a log record, denoting the record's type, is read and sequentially compared against a list of valid record types. If a match is found, the semantic routine associated with the record type is called to parse and analyze the remainder of the record. This allows new semantic routines that perform different types of analyses to be introduced without changing the basic parser. Uninteresting types of log records were given "null" semantic routines that simply skipped to the end of the record.

Initially, a routine that incremented a counter associated with the particular record type was used as the semantic routine for all log records. The resulting counts were then used to arrange the list of valid record types by their frequency of occurrence. The performance improvements accrued from this reorganization were much appreciated since parsing the complete 20 megabytes of log data took several hours on a Dorado personal computer.

[1] Often a certain activity, such as the delivery of a message, generates several log records. Sunday at Midnight, a time of low network activity, was chosen as the cutoff point to help minimize the chance of discarding a subset of related log records.

6.2 Locality of Reference

6.2.1 Methodology

A system's locality of reference was defined in Chapter 5 as "the degree to which local name servers are accessed more frequently than distant servers." In Grapevine, "local" can be interpreted as belonging to the same registry since registries correspond to geographical divisions. Localities of interests can be ascertained with a matrix that is indexed in both dimensions by registry names; rows of the matrix indicate the fraction of name service operations requested by members of the row's registry concerning names in the columns' registries. A diagonal matrix would suggest strong locality of reference.

If Grapevine logged all name service operations, then a *locality matrix* could be easily constructed from the collected log data. Unfortunately, to conserve space in the log file, Grapevine does not record name service lookups. Thus, a different strategy was needed: *measures of the locality of reference in Grapevine were obtained indirectly by observing the electronic mail traffic within and between registries*. Although, this does not account for all clients of Grapevine's registration service, the mail service is by far that largest client.

Grapevine's log data includes records of many of the events occurring in the delivery of an electronic message. Figure 6.2 depicts the log records written at various stages in the delivery process. Each message, upon creation, is assigned a unique identifier called a *postmark* [Birrell *et al.* 82]. The first log record written concerning a particular message indicates the message's postmark and its sender. When the complete message as been accepted for delivery by a Grapevine server, it is deposited in an input queue, and the number of explicit recipients (before distribution list expansion) is logged along with the message's size. After the mailboxes for all recipients have been located, two log records are generated: one categorizes the recipients as being local, remote, bad, and so on, while the second log record enumerates the names of the recipients. The message is then placed on forwarding queues to be sent to the proper servers for remote recipients. Eventually, a recipient reads his mail, including the forwarded message, and an indication of how many messages were retrieved by the user is placed in the log.

From the collected Grapevine log data, the "Created" and "RecipientLog" records were used to construct the desired locality matrix. For each of a message's recipients, a name service lookup must be performed to determine where the recipient's mailboxes reside; these lookups are directly attributable to the message's sender. Thus, the sender and list of recipients for every message are sufficient to generate a locality matrix for Grapevine mail traffic. In this matrix, the name server lookups resulting from processing mail messages are accumulated according to the registries of the mail recipients.

Two passes over the log data were required to build the locality matrix. The first pass parsed the "Created" log records and built up a stable BTree whose keys were message postmarks; the data associated with a key consisted solely of the registry of the message's creator. Note that the records concerning a given message may be dispersed in the log files and may even be generated by different servers, but they can be correlated by the message's postmark. The second pass read all of the "RecipientLog" records and used the BTree, containing about 25,000 entries, to determine a message's sender.

Only individual recipients, not the names of distribution lists, were counted in constructing the locality matrix; although distribution lists must reside is some registry, they often do not exhibit the geographical significance that individuals do. Conceptually, a message's recipients can be viewed as a tree in which internal nodes represent distribution lists that get expanded into other distribution lists or individuals. The leafs of the *recipient tree*, the individuals, were used to judge the observed locality.

This methodology is sound except for a major deficiency in the logs maintained by the Grapevine system: the complete list of recipients may not be kept in the log file. Grapevine confines "Re-

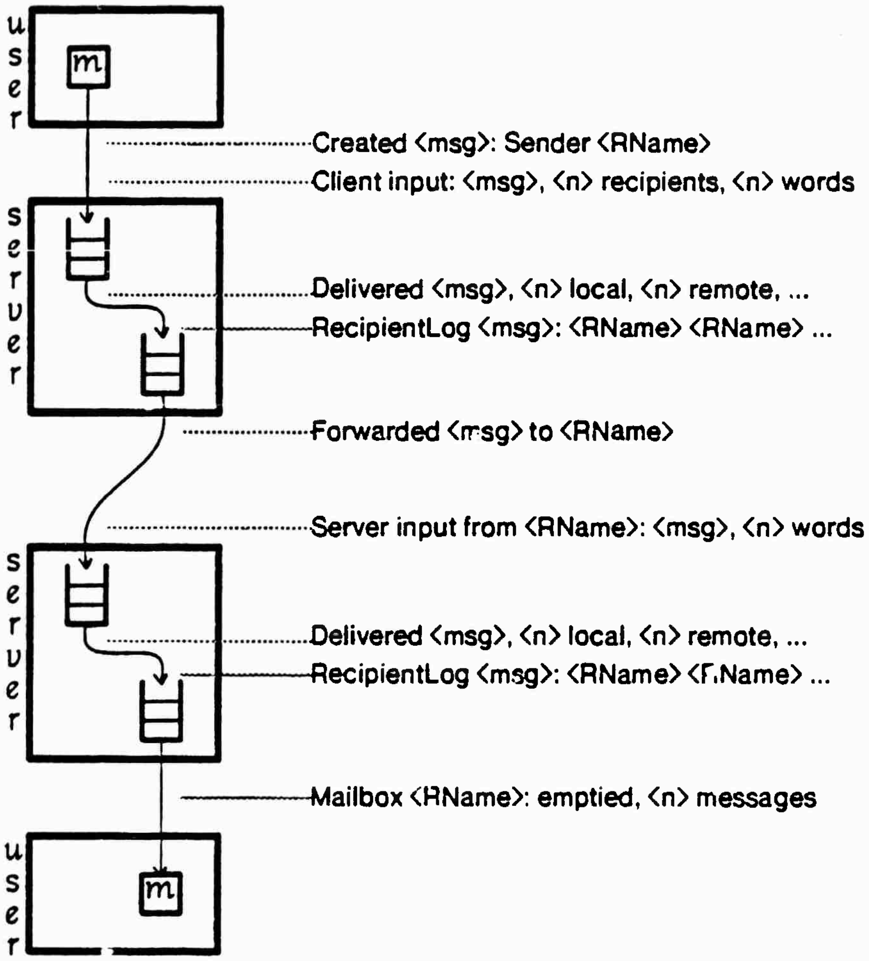

user
m
Created <msg>: Sender <RName>
Client input: <msg>, <n> recipients, <n> words
server
Delivered <msg>, <n> local, <n> remote, ...
RecipientLog <msg>: <RName> <RName> ...
Forwarded <msg> to <RName>
Server input from <RName>: <msg>, <n> words
server
Delivered <msg>, <n> local, <n> remote, ...
RecipientLog <msg>: <RName> <RName> ...
Mailbox <RName>: emptied, <n> messages
user
m

Figure 6.2: **Logging during mail delivery in Grapevine.**

	Recipients									
Sender	PA	ES	Wbst	Henr	Dlos	Pasa	Sthq	Rx	other	ARPA
PA	65	16	5	3	2	4	0	2	1	2
ES	24	51	8	7	2	3	0	3	2	0
Wbst	22	32	28	9	3	3	1	1	1	0
Henr	21	29	13	28	2	3	0	1	3	0
Dlos	27	33	8	5	15	7	2	2	1	0
Pasa	22	8	4	2	1	59	0	1	2	1
Sthq	11	9	8	0	1	0	70	0	1	0
Rx	33	21	2	2	1	1	0	37	3	0
other	29	15	4	3	2	6	0	1	39	1
ARPA	32	33	14	10	3	5	0	1	2	0

Table 6.1: **Locality of interests in Grapevine (normalised by sender).**

Entries are percentages of mail traffic normalized so that rows sum to 100.

cipientLog" records to fit in 150 characters. That is, about ten recipient names are logged on the average; when the "RecipientLog" record fills up, remaining recipient names are discarded. Thus, for large distribution lists, only the first several members are recorded. Statistically, since distribution lists are sorted by user name and not by registry name, one can argue that the initial subset of recipients characterizes the composition by registry of the complete list. However, in building the locality matrix, the truncation of recipients serves to decrease the influence of large distribution lists. Whether messages sent to large numbers of individuals exhibit different localities than those sent to a few recipients is difficult to conjecture.

6.2.2 Results

Table 6.1 gives the percentage of lookups directed to various registries as a result of mail sent from a specific registry, listed down the left-hand side. Only registries with more than 100 individuals are listed, with the others being grouped together under "other". Mail traffic to and from the Arpanet gateway is indicated under the heading "ARPA".

The diagonal of the table indicates the amount of observed locality. For example, 65% of the messages originating in the Palo Alto registry (PA) are directed to recipients in the same registry. This means that 65% of the name server lookups needed to locate the mail recipients can be performed locally if the "PA" registry is maintained close to its members. The same results normalized by recipient instead of by sender, so that the columns sum to 100% instead of the rows, are presented in Table 6.2.

For the expected high degree of locality in mail traffic, the diagonal of Tables 6.1 and 6.2 would dominate. In reality, the numbers show that the "PA" and "ES" registries participate heavily in the message traffic of all registries. For instance, of all the messages originating in "Dlos", only 15% remains in "Dlos" while 27% and 33% is destined for "PA" and "ES", respectively. The "Wbst" and "Henr" registries are in a similar situation. This implies that the authoritative name servers for the "PA" and "ES" registries receive a lot of non-local lookup requests unless these registries are freely replicated. Due to the geographical significance of registries in the Grapevine system, naming authorities for the other registries can be easily located close to their main clients, the members of the particular registry.

The measures of locality presented in the preceding two tables clearly confirm the suspicion that references to objects in large distributed computing environments do exhibit localities of interest. Nevertheless, they are specific to the Grapevine configuration and may not be valid for other dis-

	Recipients									
Sender	PA	ES	Wbst	Henr	Dlos	Pasa	Sthq	Rx	other	ARPA
PA	54	17	17	13	22	23	15	32	13	73
ES	17	45	22	27	25	13	10	32	25	13
Wbst	5	9	24	11	9	5	8	3	7	5
Henr	2	4	6	17	4	3	1	2	4	2
Dlos	2	3	2	2	14	3	4	2	3	1
Pasa	2	1	1	1	2	35	0	2	4	4
Sthq	0	0	1	0	0	0	55	0	1	0
Rx	1	1	0	0	1	0	0	23	3	0
other	1	1	0	0	1	1	1	0	28	2
ARPA	15	20	27	28	23	17	6	4	13	0

Table 6.2: **Locality of interests in Grapevine (normalised by recipients).**

Entries are percentages of mail traffic normalized so that columns sum to 100.

	Recipients							
Sender	PA	ES	Wbst	Henr	Dlos	Pasa	Sthq	Rx
PA	42	8	6	5	4	15	2	17
ES	16	24	10	12	5	11	2	20
Wbst	13	14	31	14	6	12	4	6
Henr	11	11	13	40	4	11	1	8
Dlos	14	12	7	7	26	19	6	9
Pasa	5	2	2	1	1	86	0	3
Sthq	2	1	3	0	0	0	93	0
Rx	7	3	1	1	1	1	0	86

Table 6.3: **Locality of interests in Grapevine (adjusted for registry size).**

Entries are percentages of mail traffic projected for registries of equal size and normalizeu by sender so that rows sum to 100.

tributed systems.

The numbers are particularly sensitive to variations in the sizes of different registries. Much of the interest in the "PA" and "ES" registries is due to their large size; both are twice as big as any other registry. The effects of these size differences can be accounted for by normalizing the amount of mail traffic so that the measurements represent the number of messages per individual instead of the number of messages per registry. Table 6.3 gives the percentages of message traffic normalized by sender once the registry sizes are factored out. It indicates the expected observed locality if all of the registries were of equal size.

Notice that in all cases, unlike in Table 6.1, the diagonal percentage exceeds all others, meaning that local traffic is always more common than remote traffic. If client references were uniformly distributed, all of the percentages would be around 12.5%. For the first five registries listed, many of the percentages are, in fact, in the 10-15 range. An obvious difference in basic traffic patterns, however, exists between the first five and last three registries. The last registries, "Pasa", "Sthq", and "Rx", exhibit much higher localities than expected. Once again, the need for measurements of real systems is reconfirmed.

The Grapevine measurements are also dependent on how individuals are assigned to registries.

If Grapevine registries were based on organizational boundaries that were independent of physical locations, that is, organizations were themselves globally distributed, then different localities would result. Moreover, how to exploit those localities to achieve gains in name service performance is unclear since clients with similar observed interests would be geographically dispersed. In general, exploiting the locality of clients' referencing behavior requires clustering clients with similar interests so that the name service data they frequently use can be made locally available, and hence, cheaply accessed.

6.3 Lookup/Update Ratio

6.3.1 Methodology

Grapevine's logs also contain enough data to study the conjecture that name service lookups are much more prevalent than updates. The mail system uses the registration service to maintain information about different types of objects: users and distribution lists. Users are identified by *individual names* while distribution lists have *group names*. Separate lookup/update ratios were obtained for these two types of names.

The number of lookups performed by the mail service was obtained by adding up the number of local and remote mailboxes reported in all of the "Delivered" log records. Recipient logs were used to separate the number of distribution list lookups from individual lookups. However, expanding a distribution list eventually results in lookups of individuals.

Updates to the Grapevine registration service are recorded in log records of type "RS op". For example, the log entries

```
14:38:11 RS op by 166#204. R-Name Newman.es: Create Individual
14:39:14 RS op by 166#204. R-Name Newman.es: Add Mailbox Gamay.ms
14:39:33 RS op by 166#204. R-Name Newman.es: Add Mailbox CheninBlanc.ms
```

might result from registering a new employee with Grapevine, while

```
8:55:52  RS op by 56#113. R-Name SportsCars↑.es: Remove Member Ferrari.es
9:41:00  RS op by 55#217. R-Name Gourmets↑.wbst: Add Member BCrocker.pa
```

indicate membership changes to existing distribution lists. The various types of update operations were tabulated from log records of this type.

6.3.2 Results

In a one week period, 531,039 lookup requests were presented to the Grapevine registration service by the mail service. Of the 531,039 total lookup requests, 528,338 concerned user names, leaving 2,701 accesses to distribution lists. Table 6.4 presents the number and variety of updates to user attributes, while Table 6.5 lists the updates to distribution lists for the week. For individual mail clients, $Prob(lookup)/Prob(update) = 528338/246 = 2147.72$. The assumption that the cost of name service lookups dominates a client's overall cost has been verified, at least in the case of mail senders and recipients. On the other hand, distribution list are updated much more frequently, $Prob(lookup)/Prob(update) = 2701/989 = 2.73$. The ratio of lookups to updates for individuals and groups differ by three orders of magnitude!

This simple study of one name service application client, a mail system, illustrates an important point: not only does the referencing behavior of clients vary from client to client, but also the operation mix depends heavily on the types of named objects. Both factors must be taken into consideration when designing and configuring a name service intended to serve a wide class of applications and maintain information about a diversity of named objects.

Operation	# per week
create individual	31
delete individual	25
add mailbox	116
remove mailbox	74
total	246

Table 6.4: **Individual updates in Grapevine.**

Operation	# per week
create group	6
delete group	2
add self	102
remove self	64
add member	395
remove member	420
total	989

Table 6.5: **Group updates in Grapevine.**

6.4 Applying the Name Server Model to Grapevine

6.4.1 Grapevine's configuration

The benefits of locality, based on the localities observed for Grapevine, can now be quantified by applying the name server model of Chapter 5 to the Grapevine configuration. Recall from Equation 5.1, the expected operation cost for a particular client is analytically modeled by

$$E(L_u) = \sum_{k=1}^{K} r_{uk} E(L_{uk}). \tag{6.22}$$

The cost of accessing a specific registry was presented in Equation 5.13 for replicated registries: if a client's main server is authoritative for the registry in question then the cost is given by Equation 5.8. That is,

$$L_{lookup\,uk} = \begin{cases} C_{main} + d_{main} & \text{if } NS_{main} \in S_k, \\ C_{main} + d_{main} + C_{min_{uk}} & \text{otherwise.} \end{cases} \tag{6.23}$$

To estimate the benefits of locality in Grapevine, the overall expected lookup costs for various clients are computed for both a uniform reference pattern and the locality of reference observed in the Grapevine environment.

Rather than obtaining actual measurements of the communication costs for the Xerox Research Internet, the modeled costs obtained in Table 5.1 for communicating between a client and a server are reused. Comparing Figure 5.2 with Figure 6.1 yields the associations between the labels used for name servers in Table 5.1 and the actual Grapevine server names.

As in previous analyses, clients are identified by the local network on which they reside. Table 6.6 indicates the registry with which clients on a particular local network are affiliated, as shown in Figure 6.1, as well as the main server for each client NS_{main}. The authoritative name servers for each registry were presented in an earlier paper by Michael Schroeder *et al.*, and are reproduced in a condensed form in Table 6.7. The authorities are classified as being either local, that is, in the locale of the registry's members, or remote. This table, along with Table 5.1, determines values for $C_{min_{uk}}$.

client	registry	main server
1	PA	C (Cabernet)
2	PA	D (Zinfandel)
3	ES	F (Mission)
4	ES	H (Merlot)
5	Pasa	K (PinotNoir)
6	RX	L (GreeneKing)
7	XRCC	M (deChaunac)
8	Sthq	N (Aurora)
9	Wbst	N (Aurora)
10	Henr	O (Muscat)
11	Henr	P (Catawba)
12	Dlos	Q (Barbera)

Table 6.6: **Associations between clients, registries, and main servers.**

	Storage sites	
registry	local	remote
PA	A,C,D,E	I,Q
ES	F,G,H,I,J	D,O
Wbst	N	G,P,Q
Henr	O,P	G
Dlos	Q	N,G
Pasa	K	J,N
Sthq	—	A,N,O
RX	L	K,N

Table 6.7: **Authoritative servers for Grapevine registries.**

	Lookup cost for name in registry							
client	PA	ES	Wbst	Henr	Dlos	Pasa	Sthq	Rx
1	13	75	130	130	130	185	20	388
2	13	13	75	75	75	130	75	333
3	75	13	20	20	20	75	130	388
4	20	13	75	75	75	20	185	333
5	333	333	388	388	388	13	498	13
6	646	646	333	646	333	333	333	13
7	646	646	333	646	333	333	333	333
8	959	959	326	959	326	326	326	326
9	333	333	13	333	13	13	13	13
10	646	13	333	13	333	333	13	333
11	646	646	13	13	333	333	333	333
12	13	333	13	333	13	333	333	333

Table 6.8: **Costs of accessing individual Grapevine registries.**

To enable comparisons with computations of name server operation costs presented in Chapter 5, the database access cost, d_i, is once again taken to be 6 times the cost of local network communication. The chosen cost of database operations has no significance on the comparisons performed in this section, however, since the number of database accesses in Grapevine is fixed. Locality reduces the expected lookup cost solely by reducing communication.

6.4.2 The benefits of Grapevine's locality

The cost of a name service lookup for a name in a given registry, based on the actual Grapevine configuration, is presented in Table 6.8 for all clients and major registries. These lookup costs were computed from Equation 6.23, along with the configuration data in Tables 6.6 and 6.7. They are independent of the clients' particular reference patterns.

Given a client's lookup costs for various registries, Equation 6.22 provides the client's overall expected lookup cost, an average of the lookup costs for individual registries weighted by their frequency of reference. Table 6.9 presents the expected lookup costs for all clients using both a uniform reference behavior and Grapevine's observed reference localities. For uniform referencing, 12.5% of the lookups are performed for each registry, that is, the expected overall cost is simply an average of the lookup costs for all registries. Table 6.1 provided the percentages of references for Grapevine's observed localities; a specific client's reference pattern was determined by its registry affiliation given in Table 6.6. An expected lookup cost was not computed for client #7 since the referencing behavior of registry "XRCC", which has less than 100 members, is unknown.

The third column of the Table 6.9 indicates the ratio of Grapevine's costs to uniform reference costs. For the first five clients, locality of reference results in over a 50% improvement in the overall expected lookup costs. Client #5's cost is particularly sensitive to its locality since the cost of accessing 6 of the 7 remote registries is about 30 times that of lookups to the local registry. Client #8 observed only a 20% improvement in performance despite its 70% access rate to its own registry: its problem lies in the lack of a local name server and the large expensive of accessing the "PA" and "ES" registries.

Client #9 loses big with its actual referencing behavior even though it exhibits more locality than client #12, which experiences a moderate improvement over uniform referencing. Table 6.8 shows that Client #9 can quickly access registries "Dlos", "Pasa", "Sthq", and "Rx"; unfortunately, these are referenced only 8% of the time collectively, while "PA" and "ES" receive 54% of Client #9's

	Reference pattern		ratio
client	uniform	Grapevine	G/u
1	133.88	48.61	.36
2	98.62	29.89	.30
3	92.62	41.92	.45
4	99.50	34.77	.35
5	294.25	134.86	.46
6	410.38	379.89	.93
7	450.38	—	—
8	563.38	449.34	.80
9	133.00	214.47	1.61
10	252.12	206.34	.82
11	331.25	348.31	1.05
12	213.00	169.67	.80

Table 6.9: **Expected lookup costs for Grapevine clients.**

lookup requests.

The two sets of clients, #10 and #11, both belong to the "Henr" registry, have the same referencing patterns, and are in identical positions in Grapevine's topology, as indicated in Figure 5.2. Nevertheless, one gains from its referencing behavior while the other loses. The major reason is that Client #10's main name server, Muscat, stores a copy of its most frequently referenced registry, "ES". Clients on network #11 must send lookup requests for this registry to Muscat over two slow communication lines.

6.4.3 The benefits of remote authorities

The expected cost of performing a name service lookup is highly dependent on what queries can be answered locally and which ones must be transmitted over slow communication lines. The previous section illustrated that simply storing a copy of a remote registry at a local server can substantially reduce a client's overall expected lookup cost. If Grapevine servers were only authoritative for a single registry, the registry governing their local area, then one would expect much higher lookup costs.

Table 6.10 is similar to Table 6.9 except that the cost for querying names in various registries are computed as if only local authorities existed for each registry, the authorities given in the "local" column of Table 6.7; name server "N", Aurora, is taken to be the authority for the "Sthq" registry. In this case, the overall expected lookup costs for uniform reference patterns always exceed those for the measured localities in Grapevine. Since the only cheaply accessed registry is the requestor's own registry, the improvements with Grapevine's referencing behavior stems from a client's higher than average frequency of references to the local registry.

6.4.4 Comparisons along two dimensions

The lookup costs presented in Tables 6.9 and 6.10 are reproduced in the form of a bar graph in Figure 6.3. For each client four overall expected lookup costs have been computed; the four results derive from two independent factors that affect the cost of name service queries: clients' referencing behavior and the assignment of authority over database objects. The reference patterns for each client are either assumed to be uniform or else set to be those exhibited by Grapevine's mail traffic. The two choices for configuring the name service are to assign local servers authority for local registries environments grow in unanticipated ways, resulting in performance irregularities. The utility of the performance model stems from its ability to detect such anomalies.

Chapter 7

Caching Name Server Data

Performance enhancements result from clients' acquiring local caches of name service data. Problems with maintaining strong cache consistency can be alleviated by treating cached information as hints. A new approach to managing caches of hints suggests maintaining a minimum level of cache accuracy, rather than maximizing the cache hit ratio, in order to guarantee performance improvements. The desired accuracy should be based on the ratio of lookup costs to the costs of detecting and recovering from invalid cache entries. Estimates of the accuracy of cache entries are computed from various types of metadata, such as the expected lifetime of an attribute tuple and its time since birth. Cache managers either employ revalidation procedures to restore entries whose accuracy falls below the desired threshold or simply discard the bad data. Replacement policies for caches with size constraints should consider the estimated accuracy of cache entries as well as their likelihood of future reference.

7.1 Cache Management

7.1.1 Caching for performance enhancements

Performing a name service operation may involve several interactions with name servers that are dispersed throughout a large internet environment. The high cost of resolving an object's name, however, can be substantially reduced if clients maintain local *caches* of recently acquired name server data that is likely to be reused in the future. A *cache* is an unauthoritative repository of object attributes. By consulting the cache before querying the name service, the initial cost of utilizing the name service can be amortized over several object references, assuming the cost of accessing cached data is significantly lower than that of normal query operations.

In Chapter 5, the expected cost of a name server query, $E(L_u)$, is formulated in Equation 5.1 as a function of the client's access patterns and the cost of retrieving information from a particular database partition. With caching, the cost becomes

$$E(C) = C_{cache} + P_{miss}E(L_u) \tag{7.24}$$

where P_{miss} is the probability that the desired information does not currently reside in the cache and C_{cache} is the cost of accessing the cache. Observe that $E(C) < E(L_u)$ if the cache hit ratio, $1 - P_{miss}$, is greater than $C_{cache}/E(L_u)$. Thus, if the cache access cost is much less than the expected cost of a name server query, then caching results in significant gains, even for low cache hit ratios.

Two main factors contribute to a cache's low access time in comparison with a typical name server query. First, since the cached data is stored physically close to the users of that data, the large delays in conversing with distant name servers are avoided. Second, the expensive name resolution process for locating an authoritative server for the named object in question is unnecessary.

Caches are unauthoritative in that they are used for performance enhancement only; the maintainer of a cache may store or discard cached object attributes freely without disrupting the basic name service. Caches can reside in fast volatile storage since the loss of cached data, in the event of a processor crash for instance, does not adversely affect the functional operation of the distributed name service.

7.1.2 Hints vs. strong consistency

If the name service database were immutable so that no existing database entries were ever modified, then caching data in a distributed environment could accrue all of the performance benefits and add no complexity to the clients. Realistically, the information about an object may change under normal operating conditions. For instance, an object may migrate to a new machine in order to balance the loads across machines or because its original processor crashed; in this case, the "InternetAddress" attribute maintained by the name service for the object should be updated to reflect its new location.

One approach to maintaining cache consistency would be for the name servers to inform caches whenever data is updated. However, this requires elaborate cooperation between servers and clients and generates lots of extraneous messages. Expecting a name server to know about all clients that may have cached data handed out by that server for very large internet environments does not seem feasible. It would be difficult for the servers to maintain reliable records of what information was cached by who. Such information needs to be maintained in stable storage so that it survives server crashes and might consume unreasonable amounts of storage space. Because of this difficulty in maintaining the validity of cached data, distributed systems designers often avoid caching.

An alternative approach is to treat the cached data as *hints*, which are not assumed to be completely accurate. Clients of a cache, must be prepared to deal with updates to the name service database that do not automatically propagate to the cache. The detection of inaccurate cache entries and subsequent recovery must be done by the applications that use the data in an application-specific way. Application level recovery is necessary since the appropriate action to take depends on the semantics of the data and how it is being used by the name service client.

Caches of hints have been advocated in the past [Clark 82] [Lampson 83]. The R* catalog manager [Lindsay 80] and the Grapevine mail service [Birrell *et al.* 82] both make extensive use of hints. Generally, hints about the location and availability of various services registered with a name service can be verified when clients attempt to make use of these services.

7.1.3 Cache accuracy

At any given point in time, each cache entry is either *invalid* or *valid* depending on whether or not the corresponding name service database entry has been modified unbeknownst to the cache manager. The *accuracy level* of a cache is defined to be the percentage of cache entries that are currently valid. This static measure of accuracy can be obtained by comparing a snapshot of a given cache with the name service database.

The percentage of cache lookups that return valid data to a client determines the *observed accuracy level*. This is a more dynamic notion of cache accuracy, but is difficult to quantify since it depends on the access patterns of clients over time. The observed accuracy level varies from client to client, whereas the static cache accuracy level remains independent of client behavior.

As with most caches, the *hit ratio* denotes the percentage of lookup requests that can be answered by cached data [Smith 82], regardless of the data's accuracy. With caches of hints, clients are perhaps more interested in the *accurate-hit ratio* obtained by multiplying the hit ratio by the accuracy level.

Both of these measures are highly dependent on client reference patterns and the cache management strategy.

7.1.4 A new approach to cache management

This chapter concentrates on techniques for managing cached data that may not be completely accurate, caches of hints. Because of the distributed nature of the system and the size of the environment, only the name servers that have authority for a piece of data are automatically notified when that data is modified. The existence of caches, which lie outside of the realm of the name service, is not known by the authoritative name servers. The individual applications or hosts that choose to cache name server data must *unilaterally* maintain the validity of that data since they do not participate in the usual name service maintenance operations.

The performance benefits obtained from a cache depend on the cost of accessing the cache, C_{cache}, the cost of detecting invalid cache entries for various client applications and types of data, C_{detect}, the cost of accessing the name service, $C_{NS} = E(L_u)$ obtained from Chapter 5, the update activity to the name service database, clients' referencing behavior, and the way in which the cache is managed. Suppose that the accuracy of the cache is expressed by the probability $P_{correct}$ and the hit ratio is given by P_{hit}; the expected cost of a name service query becomes

$$E(C) = C_{cache} + (1 - P_{hit})C_{NS} + (P_{hit})(1 - P_{correct})(C_{detect} + C_{NS}) \qquad (7.25)$$

where C_{detect} depends on the particular application. The cache management algorithm must determine what information should be maintained in the cache and what should be discarded so as to maximize the benefit of the cache to its clients.

Current cache memories for modern computer systems attempt to maximize the hit ratio for a fixed-size cache by utilizing intelligent cache replacement algorithms [Smith 82]. Many distributed systems that cache hints, such as Grapevine or R*, allow the size of the cache to grow indefinitely (by storing it on secondary storage); entries are only purged from the cache when detected invalid. Essentially, these systems also maximize the cache hit ratio. However, this simple scheme, which ignores the cache accuracy, may not be optimal, and may perform quite badly for data that changes frequently.

As a demonstration of why maximizing the hit ratio, or even the accurate-hit ratio, is suboptimal, suppose one cache experiences a hit ratio of P_{hit} while a second maintains a slightly higher hit ratio of $P_{hit} + \epsilon$. Assume that both caches have the same accuracy level (though, in reality, the accuracy level is probably a decreasing function of the hit ratio for a variable-size cache). The client observing the higher hit ratio gets a lower lookup cost if

$$E(C_2) < E(C_1)$$

$$\Longrightarrow \; -\epsilon C_{NS} + \epsilon(1 - P_{correct})(C_{detect} + C_{NS}) < 0$$

$$\Longrightarrow \; -C_{NS} + (1 - P_{correct})C_{detect} + (1 - P_{correct})C_{NS} < 0$$

$$\Longrightarrow \; C_{detect} < \frac{P_{correct}C_{NS}}{1 - P_{correct}}.$$

In other words, increasing the hit ratio increases the amount of invalid data returned to a client as well as improving the accurate-hit ratio. Thus, whether benefits are obtained from higher hit ratios depends on the cost of recovering from invalid data relative to the cost of straight name service lookups.

Optimal cache management involves maintaining a level of cache accuracy and a hit ratio that maximizes the benefit of the cache to its clients. Optimizing Equation 7.25, however, is difficult since the two variables, P_{hit} and $P_{correct}$, are not independent. For a variable-size cache in which only the most accurate information is retained, they are related through the size of the cache: a higher accuracy results in a smaller cache which results in a smaller hit ratio; unfortunately, the relation can not be easily quantified.

This chapter proposes a new approach to caching hints that guarantees a performance benefit from the cache, but does not attempt to derive an optimal management strategy. The agent managing the cache simply maintains a minimum level of cache accuracy. Initially, the size of the cache is limited only by the desired accuracy level. The minimum level of cache accuracy can be derived by observing that, at the very least, the cost incurred by using cached data should be less than the cost of retrieving the data directly from the name service. That is, $E(C)$ should be less than C_{NS},

$$C_{NS} > C_{cache} + (1 - P_{hit})C_{NS} + (P_{hit})(1 - P_{correct})(C_{detect} + C_{NS})$$

Assuming the cost of accessing the cache is negligible compared to the cost of a name service lookup, $C_{cache} \ll C_{NS}$

$$\Longrightarrow C_{NS} - (1 - P_{hit})C_{NS} > P_{hit}(1 - P_{correct})(C_{detect} + C_{NS})$$

$$\Longrightarrow P_{hit}C_{NS} > P_{hit}(1 - P_{correct})(C_{detect} + C_{NS})$$

$$\Longrightarrow \frac{C_{NS}}{C_{detect} + C_{NS}} > 1 - P_{correct}$$

$$\Longrightarrow P_{correct} > 1 - \frac{C_{NS}}{C_{detect} + C_{NS}}$$

$$\Longrightarrow P_{correct} > \frac{C_{detect}}{C_{detect} + C_{NS}}$$

This inequality therefore gives a lower bound for the desired cache accuracy. Generally speaking, the level of accuracy should be based on the cost of recovering from invalid cache data to achieve a successful cache management policy. If the detection cost is substantial, then the cache manager should make an effort to keep a high level of cache accuracy.

In practice, the actual static cache accuracy can not be measured since the cache manager is unaware of the state of the name service database. Instead, Section 7.4 presents techniques for estimating the accuracy of particular cache entries based on information about the lifetime of named objects. To maintain the desired accuracy level, cached data that is suspected of being invalid should be either purged or revalidated[1]. Section 7.3 examines general techniques for revalidation of cache entries. The next section discusses mechanisms for using and caching name service data in more detail.

7.2 Basics of Caching Hints

7.2.1 The cache manager

The agent responsible for maintaining the data stored in a cache is called the *cache manager*. The cache manager decides what data to keep in the cache and what data to throw away. It also responds to cache lookup requests initiated by users of the cache. Usually, name agents fill the role of cache manager, or at least call directly on cache managers as in Figure 7.1. The client base of a name agent, whether each name agent serves a single client or several clients, affects the caching strategy.

A *per-process* caching scheme, in which name server clients maintain individual caches, gives each name server client maximum control over what information it wants to retain for future use. However,

[1] In some cases, the cache accuracy that naturally results from maximizing the hit ratio may be high enough that cache entries are detected invalid and purged before they ever become suspicious. Experimental studies of existing environments are needed to determine how often these cases arise in practice.

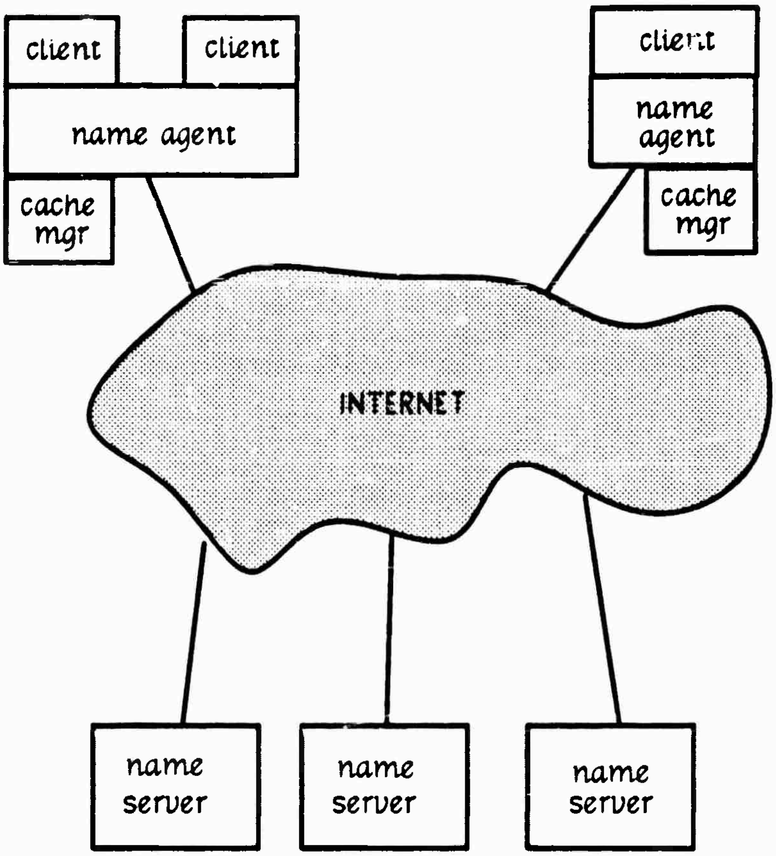


Figure 7.1: Cache managers and name agents.

having each process be the cache manager for its private cache prohibits sharing of information among processes.

To encourage sharing, it may be better to utilize a *per-processor* cache that can be accessed by all processes running on a particular machine. Due to geographical localities of interest, name server data of interest to one client may likely be of interest to another client on the same machine. A per-processor cache could eliminate duplicate name server queries issued by different clients concerning the same object.

The same arguments apply to a *per-site* cache, serving clients connected by a local area network. Communication over high-speed local networks makes such an arrangement feasible since their bandwidth may be comparable to that of a local disk. Moreover, as demonstrated in the Chapter 5, the cost of communication over a local network may be a few orders of magnitude less than the cost of resolving names in a large environment.

One can also imagine schemes in which the name servers themselves cache data returned by other servers. This cached data would then be available to all name server clients, but primarily of use to clients in the proximity of the particular server. Note that a name server assumes no responsibility or authority for data resident in a local cache. Its role as a cache manager remains distinct from its role as a name server; the first is strictly for performance reasons, while the second is a critical part of the distributed name service.

7.2.2 A cache interface

A cache can be thought of as a locally stored database object,

```
cache: Database.DatabaseObject;
```

except a cache exists outside of the distributed name service. A cache manager can utilize the database facilities proposed in Chapter 3 for storing cached data. Figure 7.2 gives a very basic interface for cache clients.

A cache read may be just a local database query, or the cache manager may attempt to check the accuracy of the cache data before returning it to the client, as discussed in Section 7.2.5. Cache writes most likely look for the existence of a cache entry with the same name and attribute value and change its value rather than naively adding a new entry to the cache. Thus, cache writes may result in either modifications or additions to the cache database. The Purge operation removes entries from the cache. As with other databases, the contents of the cache can also be enumerated. Lastly, clients, upon retrieving data from a local cache and discovering it invalid, should provide feedback to the cache manager by issuing complaints. The cache manager might choose to remove data that is known to be bad from the cache so that other clients do not ncounter the same bad data.

7.2.3 Obtaining cached data

Cached data is generally obtained indirectly from name service queries executed on behalf of clients by name agents. For instance, the query

```
value ← NS.Lookup[name, attribute];
```

may result in the tuple [name.attribute.value] being retained in the cache for future use. Name agents might simply write all data they receive from the name service to their cache, or some criteria for deciding when to cache name server data might be desirable. Name agents also receive data that can be cached from clients performing name service updates.

```
Cache: DEFINITIONS IMPORTS NS, Database = BEGIN

Read: PROCEDURE[name: NS.Name, attribute: NS.AttributeType]
    RETURNS[Database.AttributeTuple];

Write: PROCEDURE[tuple: Database.AttributeTuple];

Purge: PROCEDURE[Database.AttributeTuple];

Enumerate: PROCEDURE[next: Database.TupleID]
    RETURNS[tuple: Database.AttributeTuple, next: Database.TupleID];

Complain: PROCEDURE[Database.AttributeTuple];
END.
```

Figure 7.2: **Cache interface.**

In addition, cache managers may decide to actively query name servers to stock their caches. This *prefetching* would require cache managers to have some idea in advance of what will be desired by clients. Under certain circumstances this knowledge could come from observing past requests issued by the client along with some knowledge of the semantics of those requests. There may be a large correlation between requests for different attributes of a given object. For instance, if a client asks the name service for the host on which a given mailbox resides, then that client will likely issue a second request for the address of the mailbox's host. Thus, the cache manager may wish to *prefetch* the host's address attribute.

Since name service clients are responsible for detecting invalid cache data, the cache manager should not store data that its clients are not capable of validating at the application level. If a client is not prepared to recover from invalid data, then it should not make use of the cache. In certain cases, however, the type of the data may make it inherently difficult to detect whether the data becomes outdated while sitting in the cache. For instance, if a member is added to a distribution list that has been cached, a person sending mail to that distribution list might not be able to tell that the cached copy is incomplete. In order to know what data can be cached and what cannot, the cache manager must know something about the semantics of the data. This information could be supplied to the name service when the attributes for an object are registered and returned to name agents with any queries concerning the named object. For example, an entry in the name service database could be flagged with a "THIS DATA NOT SUITABLE FOR CACHING" warning.

7.2.4 Using cached data

A cache manager provides a subset of the name service database that can be accessed cheaply by clients indirectly through name agents. Name service clients need not even be aware of the existence of a cache as long as they treat all name service data as hints; a client's name agent's interface need not change, though, as described in Section 7.6.2, some interface changes may be desirable for dealing with the cache.

The basic name agent lookup routine for dealing with caches might be implemented as follows:

```
Lookup: PROCEDURE[name: Name, attribute: AttributeType]
RETURNS[AttributeValue] = BEGIN
    value: AttributeValue;
    cachedtuple: Database.AttributeTuple;
    cachedtuple ← Cache.Read[name, attribute];
    IF cachedtuple.value = NILTHEN
        value ← NS.Lookup[name, attribute] AT mainServerAddress
    ELSE
        value ← cachedtuple.value;
    RETURN[value];
    END;
```

When presented with a request to lookup a given name, the name agent first consults its local cache manager and returns the desired data if available; if the data is not found in the cache, then the query is forwarded to an available name server and the usual name resolution process takes place.

7.2.5 Policies for managing cached data

The basic function of a cache manager is deciding when data should be purged from the cache in order to maintain a certain level of cache accuracy. Generally, data that has been cached should remain in the cache until the cache entry is known to be invalid or suspected of being invalid. Cached data is considered ***suspicious*** when the probability that the data is valid, as estimated by the cache manager, falls below the desired level of accuracy of the cache. A cache manager must perform some action when data becomes suspect, such as purging or revalidating the suspicious cache data.

A particular cache management policy can be characterized by *when* data is checked for suspicion and *what* action is taken on suspicious data. Three approaches to discovering suspicious data can be classified as:

Passive. a cache entry becomes suspect when a name service client issues a complaint concerning the data.

OnDemand. the accuracy of a cache entry is checked when a name service client expresses an interest in that entry.

Periodic. the accuracy of cache entries is checked regularly.

In addition, three actions could be taken on suspicious data:

Purge. suspicious cache entries are simply deleted from the cache.

Refresh. new values for suspicious cache entries are obtained from an authoritative name server.

Revalidate. suspicious cache entries are checked against an authority for their validity.

One obtains a policy for cache management by combining an approach to discovering suspicious data with an action to be performed on such data. In practice, several policies may be utilized concurrently.

Passive cache management relies on clients of the cache providing feedback to the cache manager, via the Cache.Complain routine, when data becomes suspicious. Generally, feedback from a client comes after the client has attempted to use the data and discovered it invalid. For example, the Grapevine and R* systems employ a *Passive-Purge* caching policy in which the detection of invalid cache entries cause them to be flushed. A *Passive-Refresh* policy might be useful in all cases since a client is likely to reissue a particular name service lookup after complaining about the data initially returned: refreshing the data represents a type of prefetching. *Passive-Revalidate* would only be used if the cache manager does not trust clients' complaints.

Active cache managemcnt, in which a cache manager actively monitors the cache accuracy, is needed in order to provide the level of accuracy desired by a cache's clients; passive cache management policies are not sufficient to guarantee a particular level of cache accuracy. *OnDemand* policies check the accuracy of a cache entry when the client issues a query that can be answered with the cached data. If the cache entry's estimated accuracy is too low, then the cached data can not be returned unless it is refreshed or revalidated. A policy based on periodic inspection of cache entries allows entries that are never referenced to be discovered, unlike *Passive* or *OnDemand* policies. A *Periodic-Purge* policy, for instance, maintains the desired level of cache accuracy by discarding suspicious entries.

Periodic-Refresh or *Periodic-Revalidate* might be especially beneficial when free computing cycles exist on the cache manager's host; the revalidation would essentially be free in this case, ignoring the added network load induced by the action. For example, personal workstations are often left idle for short periods of time such as during lunch, coffee breaks, or phone calls. A machine might also have excess computing power during off hours. These times could be used by a cache manager to bring cache entries up-to-date.

Suppose a procedure exists to estimate the accuracy of a particular cache entry, as presented in Section 7.4,

```
AccuracyLevel: TYPE = INTEGER[0..100];

Accuracy: PROCEDURE[tuple: Database.AttributeTuple] RETURNS[AccuracyLevel];
```

as well as a general technique for revalidating cache entries, such as the ones described in Section 7.3,

```
Validate: PROCEDURE[tuple: Database.AttributeTuple] RETURNS[BOOLEAN];
```

A cache management algorithm based on either *Periodic-Revalidate* or *Periodic-Purge* might be embodied as:

```
ManageCache: PROCEDURE[desiredAccuracy: AccuracyLevel] = BEGIN
    interval: Time ← 10; -- some interval of time
    tuple: Database.AttributeTupie;
    next: Database.TupleID;
    valid: BOOLEAN;
    DO -- forever
        Sleep[interval];
        next ← NIL;
        DO
            [tuple, next] ← Cache.Enumerate[next];
            IF tuple = NIL THEN EXIT;
            IF Accuracy[tuple] < desiredAccuracy THEN BEGIN
                IF activeRevalidation THEN BEGIN
                    valid ← Validate[tuple];
                    IF valid THEN LOOP;
                    END
                Cache.Purge[tuple];
                END;
            ENDLOOP;
        ENDLOOP;
    END;
```

The parameter of this routine specifies the desired cache accuracy level.

The proper cache maintenance policies to adopt depend upon the cost of executing them versus the expected benefit. All cache managers should certainly respond to clients' complaints. In general, the cost of *Periodic* and *OnDemand* policies is difficult to quantify since one must consider the amount of spare computing cycles and other nebulous factors. Purging cache entries is quite cheap, but the cost-benefit of other approaches depends heavily on the relative cost of refresh or revalidation compared to the cost of detecting invalid data at the client level and recovering from such data. The next section discusses general techniques for refreshing and revalidating cache entries and speculates on the viability of policies employing these techniques.

7.3 Refresh/Revalidation Techniques

7.3.1 Requery strategies

The simplest way to refresh a cached attribute tuple is to reissue the query that produced the cached entry. If the requery is as costly as the original query, however, then *Refresh* policies are not cost effective in most situations. Fortunately, the cost of a second query to the name service can be substantially reduced by avoiding the name resolution mechanisms. Given the architecture developed in previous chapters, a cache manager need only keep, along with the cached data, an indication of the authoritative name server that handed out the data; subsequent queries for this data could then be sent directly to the server that has authority for the data. Name service configuration data is safe to cache since it rarely changes, and outdated configuration data can be readily detected. The cost of refreshing a cache entry has thus been reduced to the cost of querying a single name server.

The cost of requerying attributes of named objects can possibly be reduced further by caching information, such as a low level pointer into the database, that enables the server's query processor to locate the data faster. The CSNET Name Server, for instance, assigns unique "registration IDs" to all database entries for mail recipients [Solomon *et al.* 82]. Even though the database is maintained in a centralized fashion, looking up entries in the CSNET database by registration ID should be much faster than the usual keyword-matching lookups. Nevertheless, local performance enhancements of this sort may have a small net effect on response times if communication costs dominate.

OnDemand-Refresh schemes typically check the accuracy of the data and refresh suspicious cache entries before returning to the calling client. With such a policy, name service clients that are not equipped to detect invalid data can request 100% accuracy, in which case refreshes are performed for all client queries. Even though the cached attribute values are never used, the cache contains hints that enable name service lookups to be executed less costly than if the cache did not exist.

As an alternative approach to *OnDemand-Refresh*, a cache manager could return cache data to clients regardless of its estimated accuracy and immediately attempt to refresh suspicious data while the client attempts to use the cached data. For data that is suspicious but actually valid, the client obtains faster response times than if it had to wait for the data to be refreshed. For invalid data, the valid data may already be retrieved from an authoritative name server by the time the client complains. Of course, the client wastes more cycles than if it simply waited for valid data in the first place.

A cache manager could also actively try to refresh cache entries by requerying name servers independent of client requests, *Periodic-Refresh*. In general, such a scheme would not be as cost effective as *OnDemand-Refresh* unless the cache manager had some knowledge of future client requests.

7.3.2 Timestamps

Techniques for revalidating cache entries, guaranteeing that the value associated with the attribute of the named object has not been modified since the data was cached, can be based on the use of *timestamps* or *version numbers*. A *timestamp* is a strictly increasing indication of when the last update was made to a part of the database. Every modification to a name service database item should increase the value of the timestamp associated with that tuple. Timestamps can be conveniently obtained from the time of a database update. A timestamp that is a simple counter is often called a *version number* since it indicates how many times the data has been modified.

If timestamps are maintained for name service information and handed out along with the response to a name service query, then the timestamp information may be stored by a cache manager and associated with each cache entry. Revalidating a cache entry is simply a matter of comparing its timestamp with one returned from an authoritative server. If the timestamps agree, then the cache entry is guaranteed to be valid; if they differ, the cached data may or may not be valid depending on the granularity of the timestamp.

The *granularity* of a timestamp represents how much of the database is covered by the timestamp. It could range from one timestamp for the complete database to a timestamp per database attribute tuple. Another reasonable alternative would be a timestamp per named object.

The finest granularity is achieved by maintaining one timestamp per name service database tuple. In this case, each cache entry has an individual timestamp. For such fine granularity timestamps, the cost of an active *Revalidate* policy is almost the same as that of a requery algorithm since a name service query is necessary to retrieve the current timestamp.

The R* catalog managers maintain a version number per catalog entry, but they do not actively revalidate cached entries [Lindsay 80]. The version numbers are used for application-level recovery by query processors, clients of the catalog. The Grapevine registration service also keeps a timestamp per database entry [Birrell *et al.* 82]. Although the Grapevine mail service does not actively revalidate its caches, other clients could easily do so since Grapevine provides a "CheckStamp" routine that takes a name and a timestamp and returns "noChange" if the given timestamp is valid [Birrell 83].

Benefits might be obtained with a timestamp mechanism if larger granularity timestamps are used. For instance, if a single timestamp is used for all of the attributes associated with a given object, then all cache entries for an object can be revalidated, or invalidated, with a single timestamp comparison. Whenever a name service update is performed, all cache data that is covered by the same timestamp as the modified data is considered invalid regardless of whether the data has actually been modified. Thus, large granularity timestamps result in pessimistic revalidation algorithms.

In the extreme, where a single timestamp exists for the complete name service database, the timestamp can be returned with all name service queries. The whole cache can then be revalidated with a single comparison whenever a cache miss causes a name service lookup. An inexpensive scheme of this sort might perform quite well if the name service database did not change very often. On the other hand, if updates occurred with some regularity, then the cache would almost always be empty.

The Pup name lookup server, for example, provides a single timestamp for its complete database [Boggs 83]. The timestamp is used to maintain consistency among the various copies of the database. Whenever updates are made, the new timestamp is broadcast, and out-of-date servers request new versions of the database. Sites in this environment could manage their caches by listening for broadcast notices advertising new timestamps.

Ideally, the granularity of a timestamp should be adjusted according to the update frequency of the data. The benefit of revalidating several cache entries simultaneously must be traded off against the probability of invalidating perfectly good data.

7.3.3 User-supplied revalidation procedures

The use of caches that are not completely accurate relies on applications being able to detect and recover from invalid cache data. Typically the methods used by name service clients to check the validity of data vary substantially according to the type of the data and how it is used by the applications, whereas the techniques discussed thus far for refreshing and revalidating cache entries do not depend on the type or semantics of the data be validated.

Suppose application level *revalidation procedures* are formalized to the point that detecting bad data is accomplished by calling a routine which returns an indication of the validity of its arguments:

```
ValidateProc: TYPE = PROCEDURE[tuple: Database.AttributeTuple]
    RETURNS[BOOLEAN];
```

A revalidation procedure could then be handed to the cache manager along with data to be cached and used to revalidate cache entries in an application specific way. The cache manager need not understand the semantics of a revalidation procedure as long as it has a standard way of invoking the routine and interpreting the return value to decide whether or not to purge the cache entry in question.

Several problems arise with call-back procedures, however. In heterogeneous network environments, the name servers may run different operating systems and programming languages than their clients, so passing executable procedures from clients to servers may be difficult. Also, the name servers may not be able to establish the proper execution environment in which to run the procedure or possess the necessary access rights.

More importantly, placing data validation under the control of a cache manager implies that verifying the validity of data must be an independent procedure. Often, however, the validation takes place as part of the client's using the name service data, and the two functions cannot be feasibly separated due to the semantics of the application. As a real world example, consider the list of phone numbers that many people keep next to their phones. These lists are essentially caches of the real information maintained by the phone company. Revalidation of a phone number occurs when the phone number is dialed and a question of the form, "Is this so-and-so?" is posed; then the conversation continues. Theoretically, a person (or his intelligent telecommunications equipment) could periodically revalidate his cache of phone numbers by dialing each one in sequence, asking the "Does so-and-so still live there?" question, and then hanging up upon receiving the answer. In practice, this would be socially unacceptable; the commonly accepted protocol for human-to-human communication would be violated.

Similarly, existing computer communication protocols for invoking services do not generally make a clean separation of function between end-to-end validation and communication. As an exception, the Simple Mail Transfer Protocol used for transmitting electronic messages in the DARPA Internet has a "verify" command to verify the existence of a user name at a particular host [Postel 82b]. New standard validation procedures should be established for other classes of objects. For instance, an accepted "who are you?" protocol to which all network services respond would allow the availability of a service at a particular network address to be easily verified.

In conclusion, the use of client-supplied revalidation procedures for managing caches of various user-defined data types presents several formidable problems in the general case. However, standard revalidation protocols for common data types, such as mailboxes or servers, could be successfully utilized.

7.4 Estimates of Cache Accuracy

7.4.1 Probabilistic algorithms

In general, since a cache manager is unaware of name service updates, measuring the cache accuracy is impossible. However, given the expected lifetime of a name service attribute tuple, *probabilistic algorithms* speculate on the accuracy of the cached data by noting the time since the data was entered in the name service database.

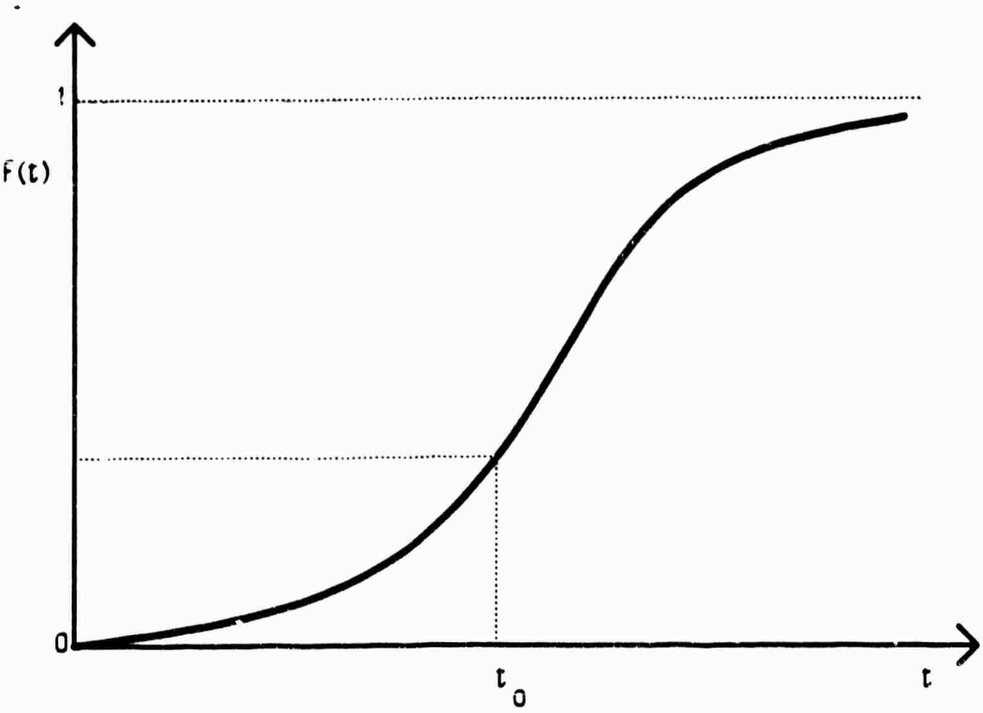


Figure 7.3: **Distribution function $F(t)$.**

The *lifetime* of an entry in the name service database is defined to be the time between successive modifications to that entry, where addition and deletion are considered to be the initial and final modifications, respectively. If name service data is considered *immutable*, that is, name service database entries are never modified, but are simply destroyed and new ones created, then the lifetime of a data item is truly the time between its birth and death.

For any reasonable lifetime distribution, except memoryless distributions, the probability that a cache entry is valid decreases with its length of time in the cache (and the time since its creation). In order for a cache manager to provide the level of accuracy desired by its clients, it must:

1. keep track of the length of time that a piece of data has been in the cache, and
2. estimate the data's accuracy, the probability that the data is still valid.

The first responsibility is simply a matter of storing a cache entry's creation time along with the data. The second function depends on how much knowledge can be obtained about the lifetime distribution of name service data.

Suppose, for a moment, that the cache manager has perfect knowledge of objects' lifetime distributions. Let L be a continuous random variable denoting the lifetime of name service database entries. Let $F(t)$ be the known distribution function of the random variable L. Then the probability that L is less than time t is given by

$$Prob(L < t) = \begin{cases} 0, & \text{if } t \leq 0, \\ F(t), & \text{if } t > 0. \end{cases}$$

Thus, if t_0 is the time since an entry was created in the name service database, $F(t_0)$ is the probability that the entry is no longer valid. Figure 7.3 depicts a sample distribution function. Observe that $F(t)$ is a nondecreasing function of t and ranges from 0 to 1. $F(\cdot)$ can be obtained from $f(t)$, the density

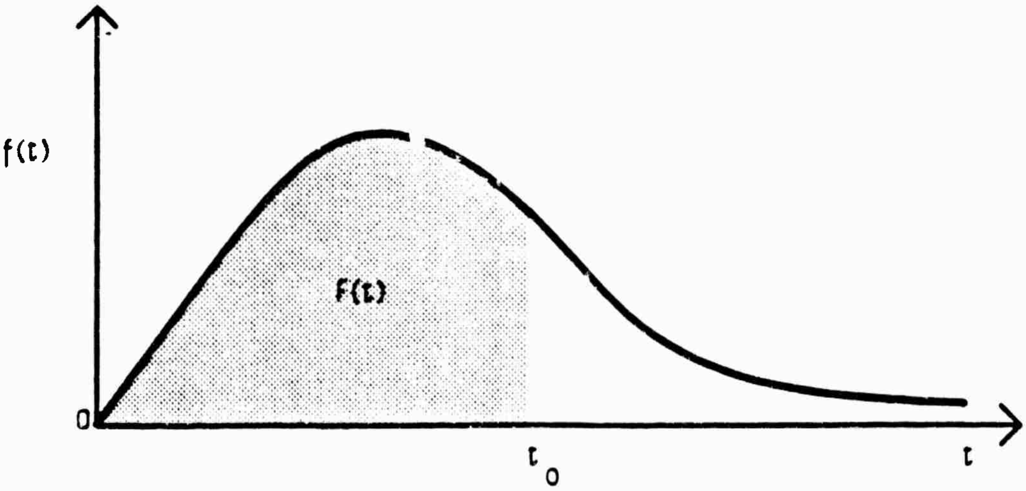


Figure 7.4: **Density function $f(t)$.**

Object	Lifetime
hosts	years
people	months-years
services	days-months
files	days
processes	minutes

Table 7.1: **Sample object lifetimes.**

function of L, by taking the area of the curve of $f(t)$ from 0 to t.as indicated in Figure 7.4. The graph of the density of L gives an intuitive feel for what values of L are likely, although, mathematically, $Prob(L = t) = 0$ for any value of t since L is a continuous random variable.

A cache's accuracy level, denoted by A, is the probability that the lifetime of a cache entry exceeds the time since the entry's creation: $Prob(L \geq t_o) > A$. Thus, in order to maintain an accuracy level A, the cache manager might discard cache entries whose time since creation exceeds time $t_{threshold}$ such that $F(t_{threshold}) = 1 - A$. The cache manager limits the *age* of the cache to the threshold time, and cached information is said to *decay* over time. The threshold value, $t_{threshold}$, represents a simple criterion for deciding how the cache should be aged, that is, when cache entries should be considered suspicious.

Realistically, one would expect that not all object attributes exhibit the same lifetime distributions. In fact a wide range of lifetimes exist. For instance, a host's internet address changes rarely, if ever, while processes come and go in a matter of minutes. In between these two extremes, lie a range of objects such as people, files, or services, and a variety of information about those objects. Table 7.1 presents a rough conjecture of the time various objects remain in a computing environment; attributes of these objects may vary more rapidly.

To obtain reasonable estimates of the accuracy of differing types of cache entries, the cache manager should maintain a table of lifetime distributions for various classes of information. Cache management would then utilize a separate threshold value per class. Identifying classes of name service attribute tuples that have similar lifetime distributions can be difficult. One heuristic would be to distinguish classes of attributes by their type and the type of the object to which they apply. For instance, the internet addresses of all hosts in the environment might have similar lifetimes. On the other hand, the functional lifetimes of files probably varies with the specific type of the file

[Satyanarayanan 81], so a single lifetime distribution for all files would yield poor estimates. In the worst case, a separate threshold value must be computed for each cache entry.

7.4.2 Estimates from imperfect knowledge

In computing the threshold for suspicious data, the previous section assumed that cache managers know a priori the lifetime distributions of the data they choose to cache. How do cache managers obtain this data? It could be obtained by observing actual behavior over a period of time, though this is not generally feasible due to the long lifetimes of many objects and the difficulty of determining a function from a limited number of sample points. Without some oracle, a cache manager must rely on name servers or clients to provide this knowledge. This section presents a series of techniques for selecting cache aging thresholds depending on how much information about an attribute's lifetime distribution is available. The initial approaches, based on very little feedback from an object's manager, probably perform unsatisfactorily in most cases.

Left on his own, with no knowledge of an attribute's lifetime, the designer of a cache management algorithm is forced to use "intuition" to pick a value for $t_{threshold}$. For instance, the Xerox Routing Information Protocol, part of the Xerox Network Systems family of communication protocols, ages caches of routing information with a seemingly arbitrary threshold time of three minutes [Xerox 31]. This time is part of the protocol specification and is independent of the network topology or other properties of the network.

The creator of an object, and the process that most likely registers it with the name service, presumably has some knowledge about how that object will be used and its expected lifetime. The object's creator cannot be expected to know the complete distribution function for the lifetime of the various attributes of the object, but should at least be able to venture a guess of the expected mean (or median) lifetime, $l_{estimate}$. This information is registered in the name service database along with the object's attributes and returned to cache managers as an aid in cache management. Such data could be used by the cache manager to set a reasonable value for the threshold time, for instance $t_{threshold} = l_{estimate}$. Of course, this ignores the level of accuracy desired by cache clients, but is better than picking an arbitrary cache age. Using an estimate of the median object lifetime as the threshold, the accuracy of the cache would be 50% since $F(median) = 1/2$. A 50% level of cache accuracy would be unsuitable for many applications.

An obvious embellishment to the simple strategy of setting $t_{threshold}$ to an estimate of the information's mean or median lifetime would be to let the threshold value vary inversely with the accuracy, given that $F(l_{estimate}) \approx 1/2$. For example, if an accuracy level of 3/4 is desired, approximately 50% more accuracy, then $t_{threshold}$ could be set to $1/2l_{estimate}$. For an arbitrary accuracy A, let $t_{threshold} = 2(1 - A)l_{estimate}$. This straight-forward interpolation of the threshold values, which takes into account the desired accuracy level A, provides better control over the cache contents without requiring any additional information about object attribute lifetimes. Statistically, it assumes that the lifetime distribution is a linear function passing through the points (0,0) and $(l_{estimate}, 0.5)$ bounded by 1; depicted in Figure 7.5. In other words, the density of attribute lifetimes is uniform over the range $(0, 2l_{estimate})$. Unfortunately, one often observes object lifetimes in practice that have a smaller variance around the mean than a uniform density.

Simple estimates for the mean object lifetime can be used more intelligently if the cache manager assumes that object lifetimes match a particular family of distributions; a gamma distribution, $F(t) = \Gamma(t; \alpha, \lambda)$, would likely be a good choice. (The density depicted in Figure 7.4 is roughly a gamma density with $\alpha = 2$.) For an assumed family of density functions with a given variance, the estimate of the mean, $l_{estimate}$, can be used to estimate the actual distribution function, $F(t)$, and subsequently derive $t_{threshold}$ given the desired accuracy level A.

Considering the gamma distribution in a bit more detail, the mean μ is given by $\mu = \alpha/\lambda$. Thus, assuming that the lifetimes of name service information are distributed according to the gamma density for a particular value of α, $F(t)$ can be approximated by $\Gamma(t; \alpha, \alpha/l_{estimate})$. Note that the

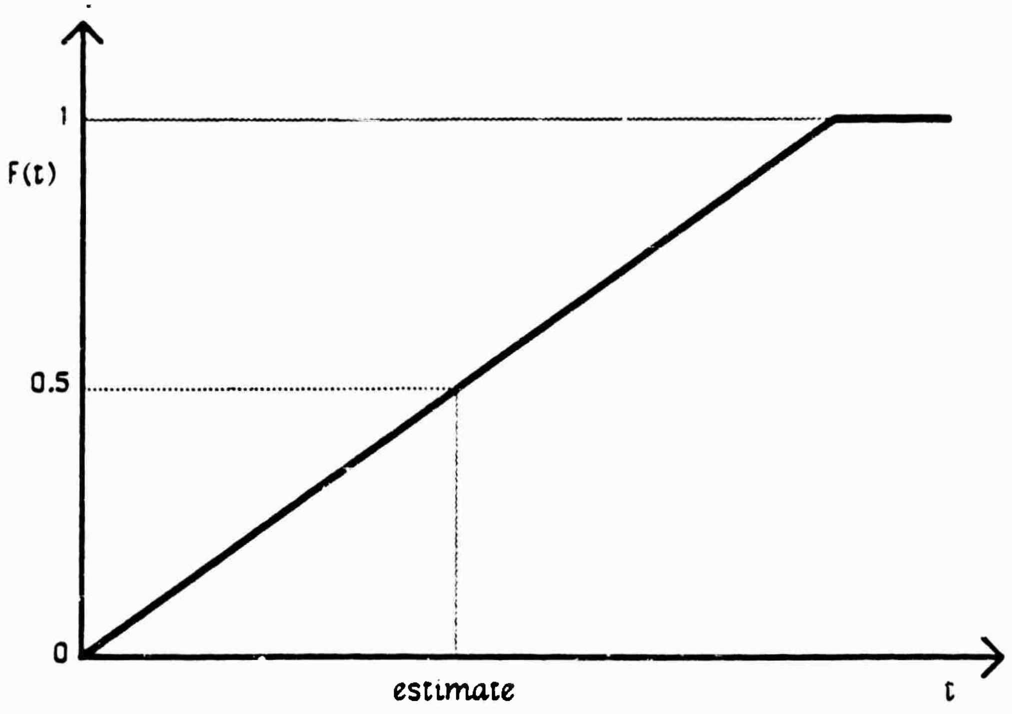


Figure 7.5: **Approximating $F(t)$ by interpolation.**

exponential family of densities is a special case of the gamma densities with $\alpha = 1$. For an exponential distribution, the rate of decay of the cache is given by $\lambda = 1/\mu$ or $\lambda = \hat{\mu}^{-1} \log 2$, where $\hat{\mu}$ represents the median.

Thus, given an estimate for the lifetime of an object attribute provided by the object's creator or manager, approximate lifetime distributions can in turn be used to derive thresholds for a given cache accuracy level. Even fairly imprecise estimates of a cache entry's accuracy permit more intelligent cache management than current cache management strategies. However, to achieve reasonably precise thresholds, and hence better performance, studies of the lifetime distributions of various objects should be conducted.

7.4.3 Accuracy with revalidation

Ways of estimating the accuracy of a cache entry, given the time since its original creation and its lifetime distribution, enable caches to be managed intelligently. When an entry is deemed to be below the desired accuracy level, one option is for the cache manager to actively revalidate the entry. Upon revalidation, the accuracy of the cache entry should get reset to 100% and start decaying again; the decay rate should be adjusted to account for the revalidation.

One approach would be to pretend that the object's creation time is the time of the revalidation. The accuracy then could be computed according to the algorithms presented in the preceding section. That is, the threshold time value, $t_{threshold}$, would still be set such that $F(t_{threshold}) = 1 - A$, though the time would be measured from the last validation point instead of the data's creation time.

While this first approach seems simple and intuitively appealing, it has a major flaw in that it assumes that the mere act of revalidating a piece of data installs new life into it. Actually, revalidation cannot affect the lifetime of the data, but merely gives added confidence in the data's continued existence. With the exception of the exponential distribution, the accuracy of the data is dependent on the time of revalidation as well as the lifetime distribution and time of creation.

Specifically, with revalidation, the accuracy of a cache entry is the probability that the data is still valid t units of time after its birth given that it was valid at the time it was last revalidated; this is known as a *conditional probability*. Suppose a particular cache entry's lifetime is represented by a continuous random variable L, and that this cache data was last known to be valid at time t_{valid} after its creation. In order to maintain a desired accuracy level A, the cache manager should considered this entry suspect $t_{threshold}$ units of time after its creation, where $Prob(L < t_{threshold} | L > t_{valid}) = 1 - A$. This conditional probability can be determined from the lifetime distribution, $F(t)$, according to Bayes' Rule,

$$Prob(L < t_{threshold} | L > t_{valid}) = \frac{Prob(L > t_{valid} \cap L < t_{threshold})}{Prob(L > t_{valid})}$$

$$= \frac{F(t_{threshold}) - F(t_{valid})}{1 - F(t_{valid})}$$

Notice that for cache entries that have never been revalidated, where $t_{valid} = 0$, this formula is simply $F(t_{threshold})$ as expected, and the algorithms presented in the previous section hold.

As discussed earlier, the common techniques for revalidating name service data are based on interactions or feedback from an authoritative name server. These assume that the authoritative name servers for an object have completely accurate information about that object. Thus, even if a cache manager makes no effort to revalidate data stored in its cache, the time of last validation, t_{valid}, should be initially set to the time that the data was retrieved from the name service. Only in cases where the name servers are not considered completely accurate is the time of last validation identical to the creation time from the cache manager's point of view. Perhaps, references to cache entries should update the time of last validation, t_{valid}, under the assumption that the client validates the data upon use and will complain to the cache manager if the data is found invalid.

7.5 Other Issues in Cache Maintenance

7.5.1 Conflicting Cache Requirements

Thus far, algorithms for maintaining a given level of cache accuracy have been discussed assuming that clients can specify the desired level of accuracy based upon the cost of recovering from inaccurate data. This section proposes a technique for managing caches that are shared by several clients with potentially conflicting requirements. For instance, one client may want very accurate data while another can easily detect and recover from invalid data. Even within the same application, different accuracies may be needed for different types of data, or for the same data used in different ways.

Suppose clients of the cache call a routine to specify the level of accuracy of the cache they wish to have maintained. Different accuracies for different classes of data can be easily accommodated. However, if several clients desire different accuracies for the same class of data, the cache manager must have some way of resolving the conflicting demands. For example, with a *Periodic-Purge* maintenance policy, choosing the lowest accuracy would be disastrous to those clients that require highly accurate data. Thus, the cache manager has little hope but to choose the highest accuracy. Unfortunately, if the clients' optimal accuracy levels differ substantially, then the client that does not need the high accuracy would experience unnecessarily low hit ratios.

To avoid these problems, the cache maintenance algorithm should not try to maintain a particular accuracy level, but should allow the accuracy to vary dynamically; so dynamically that each client perceives the cache as being at its desired level of accuracy. Suppose the cache manager never discards cache entries. (In practice, a *Periodic-Purge* algorithm can be used to delete cache entries whose accuracy falls below a minimum accuracy level.) The accuracy of cache entries can be determined by the methods described earlier, but no threshold values are computed or used to age the cache. Instead, clients specify the desired accuracy with each lookup operation. Rather than simply returning the data if found in the cache, the cache manager first checks if the data meets the accuracy requirements. If the data falls below the desired accuracy level then the cache manager pretends that the cache entry does not exist and directs the lookup to an authoritative name server. The new lookup serves to revalidate the cache entry.

Essentially, cache managers for clients with conflicting cache requirements should use *OnDemand-Refresh* with the accuracy level supplied on each lookup. This approach easily accommodates clients with different, or even changing, accuracy requirements, but may require lots of storage. For caches with real size constraints, the techniques in the next section may be used.

7.5.2 Size constraints

The size of a cache is strongly correlated to its accuracy, the access patterns of its clients, and the cache management algorithm employed, though this correlation is difficult to quantify, just as the cache hit ratio is difficult to quantify. Assuming client accesses are reasonably regular and a constant accuracy level is maintained, some steady state cache size exists. To see this, suppose the cache size at time t is S_t. At time $t + \Delta$, the cache size $S_{t+\Delta}$ is S_t minus the number of entries that have decayed plus the number of new entries added. The number of purged entries per unit time increases with the size of the cache, while the number of new entries decreases as the cache grows since new entries are only added when actual name service queries are performed, that is, when a cache miss occurs. Thus, steady state occurs when the rate of decay equals the rate of new cache acquisitions.

If the cache is maintained on plentiful disk storage, then typically the cache growth is solely dictated by the desired accuracy level. However, occasionally additional size constraints may be imposed that force the cache manager to discard data even if it meets the desired accuracy. For instance, the cache may reside in a very fast, but limited size, memory; or on a personal computer with severe disk limitations. In these cases, the cache manager must apply some criterion for deciding what entries to discard.

The many alternatives for managing fixed-size caches include:

- only add new cache entries when room exists in the cache,
- randomly replace existing cache entries with new ones,
- discard the least accurate cache entries,
- use some other measure of caching desirability.

The first alternative manages the cache as usual, but simply ignores any attempts to add data to a full cache. Thus, the accuracy of the cache remains at the desired level; the specified accuracy determines the rate of turnover in the cache. Realistically, recently acquired data that is suitable for caching is more likely to be reaccessed than old cache data, so this first alternative is probably less beneficial than schemes that choose an existing cache entry to replace.

Random cache replacement is easy to implement, but ignores the cache accuracy level. For a fixed number of cache entries, higher performance can be achieved by maintaining more accurate data.

Discarding the least accurate cache entries essentially adjusts the accuracy of the cache dynamically until the size constraints are met. However, more computation than usual is required on the part of the cache manager. The cache manager can no longer simply compute a cache entry's threshold value $t_{threshold}$ once, store it along with the entry, and check it periodically; the decision of when to discard a cache entry is based on the current estimate of its accuracy, which changes over time and hence must be periodically recomputed. A simple approximation to this algorithm could discard the cache entry nearest death, based upon the threshold value computed from the desired level of accuracy. That is, pick a victim such that the difference between its threshold time and the time it has already lived is minimized.

Maximizing the cache accuracy may not yield the optimal performance. With fixed-size caches, the decision to keep one cache entry often displaces one or more other entries, thus affecting the cache hit ratio. Neither of the two algorithms outlined above make an effort to improve the hit ratio by retaining entries most likely to be needed by clients in the near future. Unfortunately, it is difficult for the cache manager to predict future client's accesses to name service data.

Similar problems are faced by fixed-partition memory management algorithms, which must decide what pages to keep in memory [Belady 66]. Algorithms using LRU replacement policies or clock algorithms, which have existed for quite some time, attempt to predict future memory referencing behavior from observing past behavior. None of the cache management algorithms discussed in this chapter have taken into account the likelihood that the cached data is actually used. Although the situation with caching is slightly different from memory management because cache entries are neither fixed-size nor completely accurate, lessons can be learned from the older discipline.

For example, a cache manager could use a *Periodic-Purge* algorithm to guarantee a level of cache accuracy, and then discard the least recently used data to meet the size constraints. This would require the cache manager to note the time of last reference along with each cache entry. A clock-like algorithm based on "use" bits might also be used effectively. Elaborate decision policies could be designed that take into account both accuracy levels and past references.

Lastly, a cache manager might use feedback from its clients or name servers to determine the desirability of caching particular database tuples. One extreme example of this, marking data that is not suitable for caching, has already been discussed. Others include indications from clients that a given attribute will be needed in the future, or perhaps is no longer of value[2]. Also, name servers might choose to maintain statistics about global referencing patterns and relay these to cache managers.

Each of these options for managing fixed-size caches have certain advantages over the others. For some the advantage is simplicity; others attempt to increase the benefit of the cache in one way or another. Choosing the best approach requires a quantitative assessment of their effect on the expected name service lookup cost. One can not expect to determine an optimal policy for fixed-size caches at least until one for unlimited-size caches can be derived.

7.6 Name Server Support for Caching

7.6.1 Metadata

While the name service may be unaware of the existence of particular caches dispersed throughout the distributed environment, it contributes to their maintenance by maintaining information about objects' attribute tuples. In particular, certain information obtained from authoritative name servers can aid cache managers in making intelligent decisions about what data should be retained in their caches. Such information is often referred to as *metadata* since it is data about the name service data and not generally of direct interest to clients of the name service.

Metadata that may be maintained by name servers falls into four basic classes:

Event. the time various events occur in the lifetime of a database tuple,

Lifetime. information about the lifetime distribution of an attribute tuple,

Version. data that enables modifications to the name service database to be easily detected,

Advice. knowledge about the desirability of caching particular data.

As evidenced in the caching algorithms presented in this chapter, *Event* and *Lifetime*-metadata enables cache managers to maintain a particular level of cache accuracy. The success of techniques

[2] An analogy to such feedback in the memory management world is the *madvise* system call available in Berkeley UNIX (4.2 BSD), which allows processes to give advice to the kernel about their expected behavior.

for estimating the accuracy of a cache entry depends on the amount of information available from the name service about the data being cached. *Version*-metadata is used by cache managers that wish to actively revalidate cache entries in a cost effective manner, while *Advice*-metadata may be useful for caches with size constraints.

For the accuracy estimation techniques described in Section 7.4, the most important *Event*-metadata is the time an object's attribute tuple is added to the name service database; this creation time allows cache managers to detect suspicious cache entry. If absolute creation times are handed out by name servers along with the response to a query, then all name servers and cache managers must have a uniform notion of time. Due to the impreciseness in the estimated cache accuracy, the servers' clocks need not be finely synchronized; nevertheless, a reasonably consistent view of time should be presented by the name service. For example, the name service may choose to present all times in Universal Coordinated Time (UTC), and name server clocks need only be accurate within a few minutes of each other.

Alternatively, rather than returning the data's creation time, name servers could return the time since creation. Cache managers would then compare the accuracy threshold time to the sum of the time between the data's creation and lookup, as returned by the name service, and the time that the data has resided in the cache. Handing out time differentials allows name servers and cache managers to maintain independent notions of time since the absolute time as viewed by the server is never seen by others.

The DARPA Domain Naming System attempts to aid cache managers by maintaining a *time-to-live* field, which indicates how much longer the data should exist before being discarded [Mockapetris 83b]. Unfortunately, the design does not suggest how the values of these fields should be chosen. Moreover, cache maintenance based on time-to-live fields does not provide cache managers any control over the accuracy of the cache. Since such an approach does not even allow a cache manager to estimate the accuracy of cache entries, it cannot be responsive to a particular client's needs and recovery costs. Thus, *time-since-birth* information, in conjunction with *Lifetime*-metadata, is preferable to *time-to-live* fields.

Other data that fits into the *Event*-metadata class includes the time a database entry was last validated. This information would be used by cache managers that actively revalidate their cache entries, name servers that wish to cache data from other servers, and *active name servers* that play an active role in maintaining accurate authoritative data.

Lifetime-metadata, needed by cache managers to gauge a cache entry's accuracy, includes information about name service database tuples, such as their expected lifetime, their lifetime distribution function, or a family of approximate lifetime distributions. Whereas *Event*-metadata contains information about actual occurrences in the life of a particular piece of name service data, *Lifetime*-metadata represents statistical information about what is expected of the data's lifetime. As such, unlike *Event*-metadata, which must be maintained for each individual attribute, the lifetime information often pertains to a generic class of data, as discussed in Section 7.4.

Thus, assuming the name service provides a way of identifying database tuples whose lifetimes are identically distributed, *Lifetime*-metadata need only be stored once for each group and not for each database entry. One likely way of grouping database attribute tuples would be by the type of object that the data pertains to and the particular attribute type. The attribute type is an explicit part of the attribute tuple, while the type of the named object is not generally known by the name service. However, the name service database could be easily augmented with this type information: in fact, object type information could prove useful at the application level in order to allow type-checked bindings through the name service.

For cache managers that actively revalidate cache entries, name servers may maintain metadata to facilitate revalidation or the detection of modifications to the name service database. This *Version*-metadata need not be understandable by cache managers and could be based on a particular name server implementation. If the cache manager is unable to interpret the metadata, then revalidation must be done by presenting the metadata to an authoritative name server. One example of *Version*-

metadata, timestamps, was presented as a convenient way of checking the validity of cached data without having to compare the actual data values. Maintaining timestamps for database entries is simply a matter of providing the storage in the database and updating the timestamp fields whenever the database tuples are updated.

Finally, *Advice*-metadata could range from dynamic statistics about client references, used for predicting future references, to indications of what data should never be cached.

Much of this metadata desired by cache managers is readily available to the name servers; the servers should simply be programmed to retain this metadata along with the data to which it refers. A name service database entry's creation date is a good example of crucial *Event*-metadata that is easy to acquire. Indications that an entry has been modified are also simply a matter of adding a timestamp field to the database entry. Other metadata, such as an object's expected lifetime or lifetime distribution, must be obtained from knowledgeable sources such as the object's creator or manager.

7.6.2 Modified interfaces

Caching algorithms require feedback from clients and metadata from name servers in order to fulfill the needs of the cache's clients. Thus, the interfaces presented in Chapter 3 for name servers and name agents must be expanded for additional information exchange.

First, the **Update** operation for adding object attributes to the name service database should be modified to include information about the object that could aid in caching the data,

```
Metadata: TYPE = STRING;

Update: PROCEDURE[op: UpdateOps, name: Name, attribute:
    AttributeType, value: AttributeValue, info: Metadata];
```

The information desired from name service clients consists primarily of *Lifetime* and *Advice*-metadata. The name agent and name server update operations continue to look identical.

The **Lookup** operations, on the hand, differ for name agents and name servers when name agents make use of caches. The name server lookup routine remains basically the same as before, except that the metadata associated with a database tuple is returned along with its value,

```
Lookup: PROCEDURE[name: Name, attribute: AttributeType]
    RETURNS[AttributeValue, Metadata];
```

This metadata not only includes the information presented with the attribute when it was registered, but also might include *Lifetime* and *Version*-metadata maintained by the server.

The name agent's interface for accessing the name service might allow clients to specify the desired accuracy level with every lookup request,

```
Lookup: PROCEDURE[name: Name, attribute: AttributeType,
    desired: AccuracyLevel] RETURNS[AttributeValue];
```

Alternatively, the name agent's lookup operation need not change if cli simply specify an overall desired accuracy level,

```
SetAccuracy: PROCEDURE[desiredAccuracy: AccuracyLevel];
```

Also, the name agent must provide clients access to the cache's complain routine,

```
Complain: PROCEDURE[name: Name, attribute: AttributeType,
    value: AttributeValue];
```

The name agent simply passes complaints and requests for desired accuracy levels directly to its local cache manager. Clients of the name service need not use the cache interface routines.

7.7 Summary

Caches are unauthoritative repositories of name service data that has been obtained as the result of name service queries. They have fast access times, and hence should improve the overall performance of queries to the name service database assuming that recently requested data is likely to be reused in the near future. To alleviate the need to maintain perfect cache consistency, cached data must be treated as hints; clients using caches should be prepared to detect and recover from misinformation.

In order to guarantee performance benefits from using a cache, the cache manager maintains a minimum level of accuracy of the cache based on clients' recovery costs. The accuracy level of the cache can be regulated and adjusted dynamically given metadata about the lifetime distribution or expected lifetime of name service database entries. Cache maintenance algorithms age the cache with a decay rate that is a function of the data's lifetime distribution and a threshold dependent on the desired accuracy level. Cache managers may choose to either purge or revalidate cache entries that fall below the accuracy level. Even caches shared by clients with conflicting or dynamically changing requirements can be managed so that the clients perceive different accuracy levels.

Chapter 8

Final Remarks

> *... if you should come upon this spot, please do not hurry on. Wait for a time, exactly under the star.*
>
> — Antoine de Saint Exupéry, *The Little Prince.*

8.1 Reflections on the Architecture

Distributed name services enable their clients to unambiguously name objects and provide facilities for accessing information about those objects. This dissertation develops a flexible architecture for building distributed name services to facilitate sharing of objects in large and diverse computing environments. The key features of this architecture are:

- Its components are viewed in terms of the object model. The facilities are layered with well defined interfaces so that changes to the algorithms employed by one component are isolated from the other components.
- Existing database management techniques for partitioned and replicated data, recovery, authorization, and query processing can be easily adopted.
- Existing communication protocols, such as remote procedure calls, can also be adopted.
- The role of name servers, which provide the basic service, is distinguished from that of name agents, which access the service on behalf of clients. Name agents often hide the distributed nature of the name service from their clients.
- The information maintained by the name service consists of two types: attribute data and configuration data. In fact, the functions of a name server can be separated into two more specialized servers, if so desired: a *database server* that stores attribute data and a *name resolution server* that stores configuration data and assumes responsibility for resolving names.
- Authorities attributes represent a simple scheme for managing authority information and are more flexible than authority assignments based on the name structure.
- Context bindings and name clustering are the key to reducing the configuration database. Names may be clustered either syntactically or algorithmically. Syntactic clustering exploits the syntactic structure of names and adequately models existing naming conventions. Non-syntactic clustering, on the other hand, enables the method for resolving names to change without changing objects' names.
- Different styles of name resolution allow the mechanism to be tailored to the division of computational power between name servers and clients, as well as to the available communication paradigms.

- Caches of name service data that exist outside the boundaries of the name service can be effectively managed given feedback from name servers. The information passed by name servers to cache managers may include *Event*. *Lifetime*, *Version*, and *Advice*-metadata.

Observe that two basic types of bindings are performed by name services:

object name ⟶ authorities ⟶ attributes.

The first binding allows the name service itself to be reconfigured; the authoritative name servers for an object may change over time. The second allows information about objects to change freely since the name service permits late binding; information about an object, such as its internet location, is retrieved as needed rather than being built into client programs.

The mechanisms developed in this dissertation can be used to name a variety of objects in a general way. However, name management need not be implemented as a single stand-alone network service. For instance, a distributed file system may have its own directory system while host names are managed by a traditional name server. Whether a single network service is utilized for all objects in the distributed computing environment, or separate naming authorities are established for different object types remains a policy decision.

8.2 Thesis Contributions

Chapter 1 identified five principal problems in providing distributed name services for very large and diverse computing environments. Solutions to these problems are now presented as contributions of the dissertation:

- *Name resolution*: This dissertation dispels the common belief that the structure of names directly dictates the resolution process. Name structure need not, but can, be exploited to reduce and distribute the configuration data used to locate an object or its attributes. Names are resolved by a chain of context bindings determined by applying a series of clustering conditions to the name space.
- *Administrative control*: The multiple administrative entities cooperatively participating in the distributed community retain control over the placement and protection of their objects and information concerning their objects. Autonomous organizations may supply their own servers or freely choose other servers to store the attributes for their objects; authorities attributes, part of the commonly managed configuration database, indicate the authoritative servers for each object.

  The separation of attribute data from configuration data serves as an important contribution since organizations can enforce required access controls on attribute data maintained on their servers, while configuration data, which is shared by all and critical to the operation of the name service, contains only information used to resolve names.
- *Overhead costs*: Factors that affect the scaling of name services for large numbers of objects include the amount of storage required in each server and the number of servers involved in various operations.

  First, the number of objects for which an individual name server has authority determines the amount of storage needed for attribute data. Because of the fine grain assignment of authorities allowed by the architecture, no lower bound exists for the number of objects managed by a server. A small amount of configuration data is required in each server to allow them to locate other servers and remote contexts. Authoritative servers for a context must maintain the complete context, though, again, general clustering conditions permit fine grain control over the size of contexts.

  Second, the amount of interactions between servers required to perform an operation need not grow with the number of servers since broadcast and other forms of random inquiry are not utilized. The authorities attribute for each object name can be located by a single readily determined resolution chain. Thus, small workstations with limited resources can serve as servers, as well as larger machines, without negatively impacting name service operations.

- *Adaptation*: As the computing environment evolves, structure-free name management allows the name service to be readily reconfigured without renaming objects. In particular, if a name server becomes overloaded, part of its responsibilities can be off loaded to a different server. New name servers can be added to meet expanding demand for services.

  Merging independently created name spaces with different naming conventions can be accomplished since the name management mechanisms ignore the structure of names. Working out name conflicts becomes solely an administrative problem (which can be alleviated by having client software expand partially qualified names into globally unambiguous names).

- *Performance*: Results obtained from an analytical model for distributed name services show that the cost of name service operations with a decentralized service need not be appreciably greater than with a centralized service (though more storage space is required for configuration data). Substantial cost benefits can be accrued through replication that depend heavily on the topology of the environment and the delegation of authority over parts of the name space. Measurements of Xerox's Grapevine registration service indicate properties of clients' reference patterns that can be exploited to enhance performance, including sizeable localities of interest.

  The cost of accessing the name service can be amortized over several object references if clients maintain local caches of recently acquired name server data that is likely to be reused in the future. A new approach to managing caches of hints demonstrates that maintaining a minimum cache accuracy level, derived from the ratio of lookup costs to the costs of detecting and recovering from invalid cache entries, guarantees performance improvements. Estimates of the accuracy of cache entries are computed from various types of metadata, such as the expected lifetime of an attribute tuple and its time since birth. Even caches shared by clients with conflicting or dynamically changing requirements can be managed so that the clients perceive different accuracy levels.

The thesis postulated in Chapter 1 follows from these research results:

> *Physically distributed, but logically centralized, name services can be provided in a general and cost effective way, even for very large, geographically dispersed computing communities.*

8.3 Areas for Future Work

Several interesting areas for future work have surfaced in the context of the research discussed herein.

Experiments on caching should illuminate the applicability of various caching policies. Statistics on the lifetime distributions of various classes of objects need to be gathered from existing environments. Also, more elaborate models of cache behavior may allow optimal cache management strategies to be derived or determined experimentally.

Managing large attributes of an object present special problems. Certain classes of attributes, such as large mail distribution lists, are inherently distributed and should be maintained by the name service in a distributed fashion. Thus, the notion of authority for such attributes must be modified.

Name completion mechanisms are needed for converting abbreviations and aliases, which are more convenient for clients, into fully qualified names, which may be quite long and awkward for large name spaces. The DARPA Domain Name System, for instance, defines a protocol for requesting name completion, but does not specify how the completion into a fully qualified name is to be done. Simple techniques for expanding abbreviations, such as adding a default prefix, may be sufficient in some cases. In other cases, clients may want to obtain the currently registered name that is most similar to an abbreviation.

Discovering unambiguous names for various objects must take place independent of the name service. That is, clients of the name service must possess knowledge of the desired object's name before contacting the name service. In practice, names can be obtained through informal communication channels, such as human word-of-mouth, electronic messages, or system documentation. In some cases, the name of an object may be "guessed". Computer systems for discovering names based on characteristics of the objects, such as keywords or descriptions, are difficult to design for distributed environments. How to build such "yellow pages" services remains an interesting research area.

Attribute-based naming conventions, in which the set of attributes for an object, or some subset thereof, serve to identify the object, represents an alternative to simple unambiguous character string names [IFIP 84]. Difficulties in resolving names arise since a given set of attributes may or may not unambiguously identify a particular object; certainly, the set is not unique. Moreover, an attribute set that is presently unambiguous may become ambiguous in the future as objects with similar attributes are created. Techniques for effectively managing such "names" remain to be explored.

Resolving attribute-based names likely requires limited searches within a global naming graph. The name resolution algorithm presented in Chapter 4 is single threaded; as soon as a clustering condition is met, the resolution proceeds to a single new context. One could imagine allowing several attribute clustering conditions to be satisfied and "fork" concurrent name resolution chains that attempt to further disambiguate an attribute-based name. This possibility presents certain problems that were not addressed in this dissertation.

Practical experience obtained from building real systems is needed to check the viability of proposals and new ideas. Systems research entails a combination of design and implementation. This dissertation describes a new, flexible design for distributed name management. Its utility in practice remains to be explored. The construction of truly large distributed name services is just over the horizon.

Glossary

This dissertation used and introduced a fair amount of terminology, much of which does not have a universally accepted meaning. For convenient reference, the definitions of these terms as they relate to naming and name servic^^s are reproduced below. Be aware that many of these terms have different meanings in a different context.

abbreviation: a short form for a name that may be used in certain circumstances as a substitute for the complete name.

active name server: a name server that plays an active role in maintaining accurate authoritative data.

active revalidation: attempts initiated by a cache manager to check the validity of cached data.

advice-metadata: data maintained about the desirability of caching particular database tuples.

alias: one of several alternative names for an object, sometimes called a nickname.

attribute: a piece of information maintained about a named object by the name service, consisting of a type and value.

attribute lifetime: the time between successive modifications to an attribute's value.

attribute tuple: the representation of an attribute stored in the name service database, consisting of an object's name along with an attribute type and value.

attribute-based naming convention: a naming convention in which the set of attributes for an object, or some subset thereof, serve to identify the object.

authoritative name server: a name server that stores information about a particular object and assumes responsibility for reliably managing that information, also known as a naming authority for the object.

authorities attribute: an attribute whose value is the list of authoritative name servers for an object.

cache: an unauthoritative repository of recently acquired name server data, generally maintained by individual applications or hosts to improve their performance.

cache accuracy level: the percentage of cache entries that are valid at a given point in time; also the probability that a particular cache entry is valid.

cache accurate-hit ratio: a dynamic measure of the percentage of valid cache entries returned to a client.

cache aging: the process of discarding cache entries whose time since creation exceeds some threshold.

cache manager: the agent responsible for maintaining the data stored in a cache.

client/server model: a model of distributed computing that classifies active objects into servers, which offer services, and clients, which make use of those services.

clustering condition: an expression that allows the name space to be conveniently partitioned into contexts, either syntactically or algorithmically; specifically, a procedure that when applied to a name yields a true or false value.

configuration data: information stored in context objects about the authoritative name servers for every named object as well as context bindings that guide the name resolution process.

context: logically, a collection of named objects under a common geographical, organizational, or political affiliation; concretely, a special database object containing configuration data.

context binding: an attribute used for name resolution whose value gives a new name to be resolved in a new context.

database server: a specialized name server that stores attribute tuples and performs name service operations, but does not participate in name resolution.

event-metadata: data maintained about the time various events occur in the lifetime of a database tuple.

explicit context: a component of a name denoting a context in which other parts of the name exist.

flat name space: names that are simply character strings exhibiting no structure.

global name: a name that is interpreted in a consistent manner by all clients and services regardless of their location in the environment or other factors, also called an absolute name.

group name: a name that has a list of names as an attribute, typically used for such things as a mail distribution list or access control list.

hierarchical name space: names consisting of two or more parts that are strictly nested, forming levels; also called a tree-structured name space.

hints: information that may not be completely accurate, but may improve the performance of applications that are able to detect and recover from invalid data; cache data, for instance.

implicit context: a naming context that is not explicitly represented in the structure of the name.

initial context: the global context that starts the name resolution chain for all objects.

Internet address: a handle used by a program for communicating with another program over a computer network via a communication protocol.

iterative name resolution: a style of name resolution in which the name agent retains control over the resolution process; a name server does its best to resolve names using only locally available configuration data and returns to the calling name agent when it can no longer continue.

lifetime-metadata: data maintained about the lifetime distribution of an attribute tuple.

locality of reference: the degree to which local name servers are accessed more frequently than distant servers.

metacontext: a special "context" context containing the authoritative name servers for all other named contexts.

name: a character string that identifies an object, generally readable by humans and of mnemonic value.

name agent: an intermediary between name servers and their clients allowing client programs to be written as if the name service were locally available.

name distribution: the assignment of authority for parts of the name space to various name servers.

name registration: the act of registering the existence of an object with the name service and guaranteeing that the object's name is unambiguous.

name resolution: the process of determining the authoritative name servers for a given object.

name resolution chain: the list of context bindings encountered in the process of resolving a name, terminated by an authorities attribute.

name resolution server: an intermediary that accepts responsibility for iteratively resolving names on behalf of dumb name agents: also name servers containing only configuration data.

name server: an active entity that provides an instance of the name service, generally in cooperation with other name servers.

name service: a network service that enables clients to name resources or objects and share information about these objects.

name service database: the set of attributes, distributed and replicated among the name servers, for the universe of named objects.

name service metadata: information about an object's attributes that can aid cach .nangers in making intelligent decisions, such as the lifetime distribution of a particular attribute; see *advice-metadata*, *event-metadata*, *lifetime-metadata*, and *version-metadata*.

name service operation: an interface routine provided by the name service, such as *Lookup* or *Update*, that allows clients to access the name service database.

name space: the set of names complying with a given naming convention.

naming convention: the set of rules adopted for naming objects, including the syntactic representation of names as well as the their semantic interpretation.

naming network: a structured name space in which objects are named by paths through a graph; contexts comprise the nodes of the graph and edges represent named relations between contexts.

object: anything that deserves a name, such as a computer, file, process, service, distribution list, computer programmer, etc.

organisationally partitioned name space: names structured such that the organizational authority for assigning names is explicitly recognized and decoupled from the authoritative name servers for those names.

passive revalidation: the invalidation of cached data based on unsolicited feedback, usually from clients of the cache.

pattern: a template against which a name is compared, ranging from a name that may simply contain wildcards, which are denoted by "*" and match any sequence of characters, to a regular expression.

physically partitioned name space: names structured so that an object's name reflects the management authority for the name, for instance "name.server".

predestinate naming convention: a naming convention, such as a naming network, in which the name left to be resolved at any point in the resolution chain is a tail component of the original name presented for resolution.

probabilistic algorithm: a method for estimating the accuracy of cached data based on the time since the data was entered in the name service database.

recursive name resolution: a style of name resolution in which name servers recursively call other servers to continue the resolution of a name that can not be fully resolved locally; the initial name server that received the operation request returns the appropriate response after the name has been resolved and the operation performed.

relative name: a name whose interpretation depends on some local state information, such as the current machine.

requery operation: active refresh based on repeating name server lookups for cached data.

revalidation procedure: a client level routine used to detect invalid cache data in an application specific way.

structure-free name management: a flexible approach to name distribution and resolution, which breaks the strong ties between the structure of names and their management.

subalias: an alias for a particular component of a name.

suspicious cache data: a cache entry whose probability of being valid, as estimated by the cache manager, falls below the desired cache accuracy level.

timestamp: a strictly increasing indication of when the last update to a part of a database was made, sometimes called a version number; may be used for revalidating cache entries.

transitive name resolution: a style of name resolution in which the name server currently processing an operation simply forwards the operation to a server that can continue its processing; an authoritative server eventually performs the desired operation and returns the result.

unambiguous name: a name that refers to at most one object.

unique name: a name that is the only name for its referent.

version-metadata: data maintained about the time of the last update to a database tuple.

Bibliography

[Abraham and Dalal 80]
S. M. Abraham and Y. K. Dalal.
Techniques for decentralized management of distributed systems.
Proceedings 20th IEEE Computer Society International Conference (COMPCON), San Francisco, California, February 1980, pages 430-436.

[Accetta 83]
M. Accetta.
Resource location protocol.
Carnegie-Mellon University, RFC 887, December 1983.

[Allen *et al.* 82]
F. W. Allen, M. E. S. Loomis, and M. V. Mannino.
The integrated dictionary/directory system.
Computing Surveys 14(2):245-275+, June 1982.

[Allman 83]
E. Allman.
SENDMAIL – An internetwork mail router.
University of California, Berkeley, draft of March 14, 1983.

[Bayer *et al.* 78]
R. Bayer, R. M. Graham, and G. Seegmüller.
Operating Systems: An Advanced Course.
Springer-Verlag, 1978.

[Belady 66]
L. A. Belady.
A study of replacement algorithms for virtual storage computers.
IBM Systems Journal 5(2):78-101, 1966.

[Birrell 83]
A. D. Birrell.
The grapevine interface.
In *Grapevine: Two Papers and a Report*, Xerox Palo Alto Research Center, Technical Report CSL-83-12, December 1983.

[Birrell and Nelson 84]
A. D. Birrell and B. J. Nelson.
Implementing remote procedure calls.
ACM Transactions on Computer Systems 2(1):39-59, February 1984.

[Birrell *et al.* 82]
A. 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.

[Bloch *et al.* 84]

J. J.˜Bloch, D. S. Daniels, and A. Z. Spector.
Weighted voting for directories: A comprehensive study.
Department of Computer Science, Carnegie-Mellon University, Technical Report CMU-CS-84-114, April 1984.

[Boggs 83]
D. R. Boggs.
Internet Broadcasting.
Ph.D. thesis, Stanford University.
Available as Xerox Palo Alto Research Center, Technical Report CSL-83-3, October 1983.

[Boggs *et al.* 80]
D. R. Boggs, J. F. Shoch, E. A. Taft, and R. M. Metcalfe.
Pup: An internetwork architecture.
IEEE Transactions on Communications COM-28(4):612-624, April 1980.

[Bremer and Drobnik 79]
I. Bremer and O. Drobnik.
Specification and validation of a protocol for decentralized directory management.
IBM Research Report RC7880, September 1979.

[Carroll 78]
J. M. Carroll.
Names and naming: An interdisciplinary review.
IBM Research Report RC7370, October 1978.

[Cerf 79]
V. Cerf.
Internet addressing and naming in a tactical environment.
DARPA/IPTO, IEN 110, August 1979.

[Cerf and Cain 83]
V. G. Cerf and E. Cain.
The DoD Internet architecture model.
Computer Networks 7(5):307-318, October 1983.

[Cheng 84]
R. F. Cheng.
Naming and Addressing in Interconnected Computer Networks.
Ph.D. thesis, University of Illinois, Technical Report UIUCDCS-R-84-1158, January 1984.

[Cheng and Liu 82]
R. F. Cheng and J. W. S. Liu.
A coherent scheme to support location-independent references in internetwork environment.
Proceedings AFIPS National Computer Conference, 1982, pages 775-784.

[Cheriton and Mann 84]
D. R. Cheriton and T. P. Mann.
Uniform access to distributed name interpretation in the V-System.
Proceedings 4th International Conference on Distributed Computing Systems, San Francisco, California, May 14-18, 1984.

[Chesley and Rom 83]
H. R. Chesley and R. Rom.
A new approach to network name management.
Proceedings IEEE INFOCOM 83, San Diego, California, April 1983, pages 31-35 .

[Chou *et al.* 83]
W. Chou, A. A. Nilsson, and S. C. Chang.
Distributed directories in internetworking environment: Strategy and performance.
Proceedings IEEE INFOCOM 83, San Diego, California, April 1983, pages 563-571.

[Clark 82]
D. D. Clark.
Name, addresses, ports, and routes.
MIT Lab for Computer Science, RFC 814, July 1982.

[Comer 83]
D. Comer.
The computer science research network CSNET: A history and status report.
Communications of the ACM 26(10):747-753, October 1983.

[Cooper 82]
E. C. Cooper.
A network name space facility.
Computer Science Division, U. C. Berkeley, October 1982.

[Curtis and Wittie 84b]
R. Curtis and L. Wittie.
Global naming in distributed systems.
IEEE Software 1(3):76-80, July 1984.

[Dalal 82]
Y. K. Dalal.
Use of multiple networks in Xerox' Network System.
Computer 15(10):82-92, October 1982.

[Dalal and Printis 81]
Y. K. Dalal and R. S. Printis.
48-bit internet and ethernet host numbers.
Proceedings 7th Data Communications Symposium, Mexico City, Mexico, October 1981, pages 240-245.

[Daniels and Spector 83]
D. Daniels and A. Z. Spector.
An algorithm for replicated directories.
Proceedings Second ACM Symposium on Principles of Distributed Computing, Montreal, Canada, August 1983.

[Deutsch 79]
D. P. Deutsch.
A suggested solution to the naming, addressing, and delivery problem for Arpanet message systems.
Network Information Center, SRI International, RFC 757, September 1979.

[Feinler 77]
E. J. Feinler.
The identification data base in a networking environment.
1977 National Telecommunications Conference Record, 1977, pages 21:3.1-3.5.

[Feinler *et al.* 82]
E. Feinler, K. Harrenstien, Z. Su, and V. White.
DoD Internet host table specification.
Network Information Center, SRI International, RFC 810, March 1, 1982.

[Garcia-Luna and Kuo 81]
J. J. Garcia-Luna and F. F. Kuo.
Addressing and directory systems for large computer mail systems.
Proceedings IFIP TC6 International Symposium on Computer Message Systems, North-Holland, Ottawa, Canada, 1981, pages 297-313.

[Gelernter 84]
D. Gelernter.
Dynamic global name spaces on network computers.

Proceedings 1984 International Conference on Parallel Processing, Columbus, Ohio, August 1984, pages 25-31.

[Gifford 79]
D. K. Gifford.
Weighted voting for replicated data.
Proceedings Seventh Symposium on Operating Systems Principles, December 1979, pages 150-162.

[Gray 78]
J. N. Gray.
Notes on database operating systems.
In Bayer *et al.* [Bayer *et al.* 78], pages 393-481.

[Harrenstien 77]
K. Harrenstien.
NAME/FINGER.
Network Information Center, SRI International, RFC 742, December 1977.

[Harrenstien and White 82]
K. Harrenstien and V. White.
NICNAME/WHOIS.
Network Information Center, SRI International, RFC 812, March 1982.

[Harrenstien *et al.* 82]
K. Harrenstien, V. White, and E. Feinler.
Hostnames server.
Network Information Center, SRI International, RFC 811, March 1982.

[Hinden *et al.* 83]
R. Hinden, J. Haverty, and A. Sheltzer.
The DARPA Internet: Interconnecting heterogeneous computer networks with gateways.
Computer 16(9):38-48, September 1983.

[Hoffman *et al.* 83]
M. Hoffman, R. Schantz, R. Thomas, and B. Woznick.
Cronus, a distributed operating system: Preliminary system/subsystem specification.
Bolt Beranek and Newman Inc., draft of June 2, 1983.

[Holler 81]
E. Holler.
Multiple copy update.
In Lampson *et al.* [Lampson *et al.* 81], pages 284-307.

[IFIP 83]
IFIP WG 6.5.
Naming and directory services for message handling systems.
Working paper, version 4, July 1983.

[IFIP 84]
IFIP WG 6.5.
A user-friendly naming convention for use in communication networks.
Working paper, version 3, March 1984.

[ISO 81]
ISO/TC97/SC16.
Data processing-Open systems interconnection-Basic reference model.
Computer Networks 5(2):81-118. April 1981.
Approved as ISO International Standard IS 7498.

[Janson *et al.* 83]
P. A. Janson, W. Bux, and E. Mumprecht.

Addressing and naming in local-area inter-networks.
IBM Zurich Research Laboratory.
Presented at workshop on Ring Technology Local Area Networks, Kent, U.K., September 1983.

[Kerr 81]
I. H. Kerr.
Interconnection of electronic mail systems - A proposal on naming, addressing and routing.
Proceedings IFIP TC6 International Symposium on Computer Message Systems,
North-Holland, Ottawa, Canada, 1981, pages 315-326.

[Lampson 81]
B. W. Lampson.
Atomic transactions.
In Lampson *et al.* [Lampson *et al.* 81], pages 246-265.

[Lampson 83]
B. W. Lampson.
Hints for computer system design.
Proceedings Ninth Symposium on Operating Systems Principles, Bretton Woods, New Hampshire, October 1983, pages 33-48.

[Lampson *et al.* 81]
B. W. Lampson, M. Paul, and H. J. Siegert, editors.
Distributed Systems - Architecture and Implementation.
Springer-Verlag, 1981.

[Landweber *et al.* 83]
L. Landweber, M. Litzkow, D. Neuhengen, and M. Solomon.
Architecture of the CSNET Name Server.
Proceedings ACM SIGCOMM '83 Symposium, Austin, Texas, March 1983, pages 146-153.

[Leach *et al.* 82]
P. J. Leach, B. L. Stumpf, J. A. Hamilton, and P. H. Levine.
UIDS As internal names in a distributed file system.
Proceedings ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing,
Ottawa, Canada, August 1982, pages 34-41.

[Lindsay 80]
B. Lindsay.
Object naming and catalog management for a distributed database manager.
Proceedings Second International Conference on Distributed Computing Systems, Paris, France, April 1981, pages 31-40.
Also available as IBM Research Report RJ2914, August 1979.

[Lindsay *et al.* 84]
B. G. Lindsay, L. M. Haas, C. Mohan, P. F. Wilms, and R. A. Yost.
Computation and communication in R*: A distributed database manager.
ACM Transactions on Computer Systems 2(1):24-38, February 1984.

[Lindsay *et al.* 79]
B. Lindsay *et al.*.
Notes on distributed databases.
IBM Research Report RJ2571, July 1979.

[Livesey 79]
J. Livesey.
Inter-process communication and naming in the Mininet system.
Compcon '79, Spring 1979, pages 222-229.

[Lyngbaek and McLeod 82]
P. Lyngbaek and D. McLeod.

A distributed name server for information objects.
Computer Science Department, University of Southern California, Technical Report TR-200, December 1982.

[Martella and Schreiber 80]
G. Martella and F. A. Schreiber.
A data dictionary for distributed databases.
Proceedings International Symposium on Distributed Data Bases, Paris, France, North-Holland, 1980, pages 17-33.

[Mills 81]
D. L. Mills.
Internet name domains.
COMSAT Laboratories, RFC 799, September 1981.

[Mitchell *et al.* 79]
J. G. Mitchell, W. Maybury, and R. Sweet.
Mesa language manual (version 5.0).
Xerox Palo Alto Research Center, Technical Report CSL-79-3, April 1979.

[Mockapetris 83a]
P. Mockapetris.
Domain names - Concepts and facilities.
USC Information Sciences Institute, RFC 882, November 1983.

[Mockapetris 83b]
P. Mockapetris.
Domain names - Implementation and specification.
USC Information Sciences Institute, RFC 883, November 1983.

[Mogul 84]
J. Mogul.
Representing information about files.
Proceedings 4th International Conference on Distributed Computing Systems, San Francisco, California, May 1984, pages 432-439.

[Needham and Herbert 82]
R. M. Needham and A. J. Herbert.
The Cambridge Distributed Computing System.
Addison-Wesley, 1982.

[Needham and Schroeder 78]
R. M. Needham and M. D. Schroeder.
Using encryption for authentication in large networks of computers.
Communications of the ACM 21(12): 993-999, December 1978.

[Nowitz 78]
D. A. Nowitz.
Uucp implementation description.
UNIX Programmer's Manual, seventh edition, volume 2, Bell Laboratories, October 1978

[Oppen and Dalal 83]
D. C. Oppen and Y. K. Dalal.
The Clearinghouse: A decentralized agent for locating named objects in a distributed environment.
ACM Transactions on Office Information Systems 1(3):230-253, July 1983.
An expanded version of this paper is available as Xerox Report OPD-T8103, October 1981.

[Pickens *et al.* 79b]
J. R. Pickens, E. J. Feinler, and J. E. Mathis.
The NIC name server-A datagram based information utility.
Proceedings 4th Berkeley Workshop on Distributed Data Management and Computer Networks, August 1979, pages 275-283.

[Popek *et al.* 81]
G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel.
LOCUS: A network transparent, high reliability distributed system.
Proceedings Eighth Symposium on Operating Systems Principles, Pacific Grove, California, December 1981, pages 169-177.

[Postel 79]
J. Postel.
Internet name server.
Information Sciences Institute, University of Southern California, IEN 116, August 1979.

[Postel 82b]
J. Postel.
Simple mail transfer protocol.
USC Information Sciences Institute, RFC 821, August 1982.

[Postel 84]
J. Postel.
Domain Name System implementation schedule - revised.
USC Information Sciences Institute, RFC 921, October 1984.
Previous schedules were released as RFC 881, November 1983, and RFC 897, February 1984.

[Postel *et al.* 81]
J. B. Postel, C. A. Sunshine, and D. Cohen.
The ARPA Internet Protocol.
Computer Networks 5(4):261-271, July 1981.

[Roberts and Wessler 70]
L. G. Roberts and B. D.Wessler.
Computer network development to achieve resource sharing.
Proceedings AFIPS Spring Joint Computer Conference, 1970, pages 543-549.

[Rosen 81]
E. C. Rosen.
Logical addressing.
Bolt Beranek and Newman Inc., IEN 183, May 1981.

[Saltzer 78]
J. H. Saltzer.
Naming and binding of objects.
In Bayer *et al.* [Bayer *et al.* 78], pages 99-208.

[Saltzer 82]
J. H. Saltzer.
On the naming and binding of network destinations.
Proceedings IFIP/TC6 International Symposium on Local Computer Networks, Florence, Italy, April 19-21, 1982, pages 311-317.

[Satyanarayanan 81]
M. Satyanarayanan.
A study of file sizes and functional lifetimes.
Proceedings Eighth Symposium on Operating Systems Principles, Pacific Grove, California, December 1981, pages 96-108.

[Schicker 82]
P. Schicker.
Naming and addressing in a computer-based mail environment.
IEEE Transactions on Communications COM-30(1):46-52, January 1982.

[Schroeder *et al.* 84]
M. D. Schroeder, A. D. Birrell, and R. M. Needham.
Experience with Grapevine: The growth of a distributed system.
ACM Transactions on Computer Systems 2(1):3-23, February 1984.

[Shoch 78]
J. F. Shoch.
Internetwork naming, addressing, and routing.
Proceedings 17th IEEE Computer Society International Conference (COMPCON), September 1978, pages 72-79.

[Sirbu and Sutherland 84]
M. A. Sirbu, Jr. and J. B. Sutherland.
Naming and directory issues in message transfer systems.
Proceedings IFIP WG-6.5 International Working Conference on Computer Message Services, Nottingham, England, May 1984.

[Smith 82]
A. J. Smith.
Cache memories.
Computing Surveys 14(3):473-530, September 1982.

[Solomon *et al.* 82]
M. Solomon, L. H. Landweber, and D. Neuhengen.
The CSNET Name Server.
Computer Networks 6(3):161-172, July 1982.

[Su 82]
Z. Su.
A distributed system for internet name service.
Network Information Center, SRI International, RFC 830, October 1982.

[Su and Postel 82]
Z. Su and J. Postel.
The domain naming convention for internet user applications.
Network Information Center, SRI International, RFC 819, August 1982.

[Sunshine 82]
C. A. Sunshine.
Addressing problems in multi-network systems.
Proceedings INFOCOM 82, Las Vegas, Nevada, March 1982, pages 12-18.

[Sunshine and Postel 80]
C. Sunshine and J. Postel.
Addressing mobile hosts in the ARPA internet environment.
USC Information Sciences Institute, IEN 135, March 1980.

[Terry 82]
D. Terry.
The COSIE Name Server.
IBM San Jose Research Lab, Internal Memo , June 1982.
Available as IBM Research Report RJ4161, January 1984.

[Terry 84]
D. B. Terry.
An analysis of naming conventions for distributed computer systems.
Proceedings ACM SIGCOMM '84, Montreal. Quebec. June 1984, pages 218-224.

[Terry *et al.* 84]
D. B. Terry, M. Painter, D. Riggle, and S. Zhou.
The Berkeley Internet Name Domain Server.
Proceedings USENIX Summer Conference, Salt Lake City, Utah, June 1984. pages 23-31.
Also available as Computer Science Division. U. C. Berkeley. Report No. UCB/CSD 84/182. May 1984.

[Thiel 83]
G. I. Thiel.

- *Partitioned Operation and Distributed Data Base Management Systems Catalogs.* Ph.D. thesis, University of California, Los Angeles, UCLA Report No. CSD-83096, August 1983.

[Thomas 73]
R. H. Thomas.
A resource sharing executive for the ARPANET.
Proceedings AFIPS National Computer Conference, 1973, pages 155-163.

[Walker *et al.* 83]
B. Walker, G. Popek, R. English, C. Kline, and G. Thiel.
The LOCUS distributed operating system.
Proceedings Ninth Symposium on Operating Systems Principles, Bretton Woods, New Hampshire, October 1983, pages 49-70.

[Watson 81]
R. W. Watson.
Identifiers (naming) in distributed systems.
In Lampson *et al.* [Lampson *et al.* 81], pages 191-210.

[Wu 83]
Y. Wu.
Performance of file directory systems on a network with redundant data bases.
Proceedings IEEE INFOCOM 83, San Diego, California, April 1983, pages 572-580.

[Xerox 81]
Xerox Corporation. Internet transport protocols.
Xerox System Integration Standard 028112, December 1981.