Technical Report

UCAM-CL-TR-270
ISSN 1476-2986

Number 270

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Designing a universal name service

Chaoying Ma

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted October 1992 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Newnham College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

ISSN 1476-2986

Designing A Universal Name Service

Chaoying Ma

Summary

Generally speaking, naming in computing systems deals with the creation of object identifiers at all levels of a system architecture and the mapping among them. Two of the main purposes of having names in computer systems are (a) to identify objects; (b) to accomplish sharing. Without naming, no computer system design can be done.

The rapid development in the technology of personal workstations and computer communication networks has placed a great number of demands on designing large computer naming systems. In this dissertation, issues of naming in large distributed computing systems are addressed. Technical aspects as well as system architecture are examined. A design of a Universal Name Service (UNS) is proposed and its prototype implementation is described. Three major issues on designing a global naming system are studied. Firstly, it is observed that none of the existing global name services provides enough flexibility in restructuring name spaces, more research has to be done. Secondly, it is observed that although using stale naming data (hints) at the application level is acceptable in most cases as long as it is detectable and recoverable, stronger naming data integrity should be maintained to provide a better guarantee of finding objects, especially when a high degree of availability is required. Finally, configuring the name service is usually done in an ad hoc manner, leading to unexpected interruptions or a great deal of human intervention when the system is reconfigured. It is necessary to make a systematic study of automatic configuration and reconfiguration of name services.

This research is based on a distributed computing model, in which a number of computers work cooperatively to provide the service. The contributions include: (a) The construction of a Global Unique Directory Identifier (GUDI) name space. Flexible name space restructuring is supported by allowing directories to be added to or removed from the GUDI name space. (b) The definition of a two-class name service infrastructure which exploits the semantics of naming. It makes the UNS replication control more robust, reliable as well as highly available. (c) The identification of two aspects in the name service configuration: one is concerned with the replication configuration, and the other is concerned with the server configuration. It is notable that previous work only studied these two aspects individually but not in combination. A distinguishing feature of the UNS is that both issues are considered at the design stage and novel methods are used to allow dynamic service configuration to be done automatically and safely.

Preface

Except where otherwise stated in the text, this dissertation is the result of my own work and is not the outcome of work done in collaboration.

I hereby declare that this dissertation is not substantially the same as any I have submitted for a degree or diploma or any other qualification at any other university.

I further state that no part of my dissertation has already been, or is being currently submitted for any such degree, diploma or other qualification.

Chaoying Ma

1 October 1992

Trademarks

UNIX is a trademark of AT&T.
DECStation, ULTRIX, and MicroVAX-II are trademarks of the Digital Equipment Corporation.
Ethernet is a trademark of XEROX.
NFS is a trademark of SUN Microsystems Inc.

Acknowledgements

I would like to thank Roger Needham, my supervisor, sincerely for his valuable advice and encouragement during my research. Also, thanks to Jean Bacon, and to members of The System Research Group for their help.

I appreciate the assistance provided by Roger Needham, Sai Lai Lo and Jean Bacon who suggested improvements to the dissertation, and to John Bates, Richard Black, Simon Crosby, Joe Dixon, Mark Hayter, Paul Jardetzky, Timothy Roscoe, and Cormac Sreenan for proof-reading the text.

Thanks to my husband Zhixue, for his tireless help, encouragement and love. Also, thanks to my parents, aunt Yingqun, and Zhihui for being so supportive.

This work was supported by the Cambridge Overseas Trust, the Cambridge Computer Laboratory and Newnham College.

Abstract

Generally speaking, naming in computing systems deals with the creation of object identifiers at all levels of a system architecture and the mapping among them. Two of the main purposes of having names in computer systems are (a) to identify objects; (b) to accomplish sharing. Without naming, no computer system design can be done.

The rapid development in the technology of personal workstations and computer communication networks has placed a great number of demands on large computer naming systems. Despite the existence of several such naming systems in a distributed system environment, there still are many unanswered questions. In this dissertation, the issues of naming in large distributed computing systems are addressed. Technical aspects as well as system architecture are examined. A design of a Universal Name Service (UNS) is proposed and its prototype implementation is described. Three major issues on designing a global naming system are studied. Firstly, it is observed that none of the existing global name services provides enough flexibility of restructuring name spaces, more research has to be done. Secondly, although using stale naming data (hints) at the application level is acceptable in most cases as long as it is detectable and recoverable, stronger naming data integrity should be maintained to provide a better guarantee of finding objects, especially when a high degree of availability is required. Finally, configuring the name service is usually done in an ad hoc manner, leading to unexpected interruptions or a great deal of human intervention when the system is reconfigured. It is necessary to make a systematic study of automatic configuration and reconfiguration of name services.

This research is based on a distributed computing model, in which a number of computers work cooperatively to provide the service. The contributions include: (a) The construction of a Global Unique Directory Identifier (GUDI) name space. Flexible name space restructuring is supported by allowing directories to be added to or removed from the GUDI name space. (b) The definition of a two-class name service infrastructure which exploits the semantics of naming. It makes the UNS replication control more robust, reliable as well as highly available. (c) The identification of two aspects in the name service configuration: one is concerned with the replication configuration, and the other is concerned with the server configuration. It is notable that previous work only studied these two aspects individually but not in combination. A distinguishing feature of the UNS is that both issues are considered at the design time and novel methods are employed to allow dynamic service configuration to be done automatically and safely.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

For many years people have relied on names to identify each other and most things. Similarly, computers use identifiers (names) to refer to objects and each other in a distributed computing environment. Naming in computer systems is concerned with accesses of various objects given their identifiers. Names can exist in many different ways and are used by many different computer-based services or applications. Although the fundamentals of computer naming have been defined and well understood, and many techniques have been developed to construct computer naming systems, there are a number of issues which warrant further studies in the design of a large, distributed naming system. In this dissertation, naming in large distributed computing systems is addressed. Technical aspects as well as the system architecture are examined. The design of a Universal Name Service (UNS) is proposed, and its prototype implementation is described.

It is generally understood that naming in computing systems deals with creating identifiers at all levels of a system architecture and mapping among them. Names are given to processes, program variables, database entries, files, machines, mailboxes, gateways, and networks in a computer system. Two of the main purposes of having names in computer systems are to identify objects and to accomplish sharing. Other purposes include security-related purposes, locating, scheduling and supporting cooperation and communication. Because of the importance of naming, it has been said [Watson 81] that:

> *Identification systems (often called naming systems) are at the heart of all computer system design.*

Naming in centralised computing systems, particularly operating systems, had been studied extensively long before the recent and rapid development in computer networking and personal workstation technologies. An excellent paper entitled "Naming and Binding of Objects" by Saltzer [Saltzer 79] sets the cornerstone of research into computer naming in the late 1970s. In Chapter 2, general naming concepts will be introduced. Many of the concepts are taken from Saltzer's paper because they are also applicable to distributed naming. This dissertation focusses on distributed naming only. Some notable contributions in this area include [Comer 87, Lampson 86, Saltzer 82, Birrell 82, Shoch 78].

With regard to the large scale distributed systems, which are made possible by the growing capability of communication networks (for instance, the ISDN [Pandhi 87]), one potential requirement of naming is to enable any end user on an interconnected network to communicate with millions of others. Sharing of resources via computer networks should also be enhanced. Naming systems have expanded rapidly not only in terms of function but also capacity. People want to have a common way of handling world wide communication [Watson 81, Lampson 86, Quarterman86, Estrin 86, Araujo 88]. In respond to this, there are already some global naming systems operating; for example, the Domain Name System (DNS) [Mockapetri88], the DECdns [Martin 89], Quipu - an X.500 pilot system [Kille 89] and Xerox Clearinghouse [Oppen 83].

However, despite the existence of several naming services in the field of distributed systems, there are still many unresolved questions remaining. This dissertation tackles three issues in designing a global naming system. Firstly, it is observed that none of the existing global name services provides enough flexibility in restructuring name spaces, more research has to be done. Secondly, although using stale naming data (hints) at the application level is acceptable in most cases, as long as it is detectable and recoverable, stronger naming data integrity should be maintained to provide a better guarantee of finding objects, especially when a high degree of availability is required. Finally, configuring a name service is usually done in an ad hoc manner, leading to unexpected interruptions or a great deal of human intervention when the system is reconfigured. It is necessary to make a systematic study of automatic configuration and reconfiguration of the name service.

1.2 Overview of Solutions

In design of a very large and highly complex distributed naming system, it is vital to give the system the ability of growth in order to meet organisational requirements. For example, several existing name spaces might have to be merged to allow their clients to

share each others' names. Rapid growth of the DNS user community has placed great strain on the system. The DNS copes with the scaling problem by allowing its naming hierarchy to extend downwards from the existing nodes. However the growth of existing systems upwards is not supported by the DNS, nor is the migration of a branch of the tree from one node to another. Similar problems exist in many other name systems. The challenge that name service designers face is how to avoid imposing unnecessary limits on a system. One elegant solution to the scaling problem is to use global unique identifiers. In "Designing a Global Name Service" [Lampson 86], global unique identifiers are used for restructuring name spaces. An example of such restructuring is to merge existing name spaces by adding a new root. In many previous designs of naming system, adding a new root to a name space requires a high degree of renaming. However, using unique identifiers, the new names in the merged name space and the old names in the original name spaces are both valid names of the system. The GNS proposes to solve this problem by making only the current roots recognisable by the system, and discarding older roots, i.e. those that existed before the last merge operation took place. In the UNS, a *global UDI name space* is defined and implemented. Flexibility in restructuring name spaces is then provided by allowing the addition of directories (not necessarily a root directory) to or the removal of directories from the global UDI name space. Hierarchical name resolution is also supported by the UNS since simply using unique identifiers does not satisfy other requirements such as autonomy. Hierarchical naming is particularly useful to avoid ambiguity.

Maintenance of a universal name service still remains an unresolved issue with respect to availability and reliability, although many replication control algorithms have been extensively developed for distributed systems [Stefano 87, Bernstein 87]. In fact, most of these algorithms are designed for distributed database management or file systems. For instance, weighted voting is employed by distributed file systems [Gifford 79, Bloch 82]; this method is not very suitable for maintenance of a large naming system since it relies on an underlying atomic action mechanism which is expensive, and unlikely to scale well. Instead, loose-coupled replication control mechanisms are widely used by existing naming systems [Birrell 82, Mockapetri88, Lampson 86, Demers 87], but few can really meet the increasing demands made on naming systems by various applications.

A notable feature of the UNS name service is its large scale. Tens of thousands of organisations are likely to be involved in such a system. It is obvious that replication must be employed to improve performance and availability. In order to maintain replicated naming data, the semantics of the UNS is exploited, and a two-class name service infrastructure is proposed. In the UNS, replicated data may be one of three types. Some are defined

as *replicas* which are managed by a small number of servers called the first class servers. Some are defined as *read-only copies* maintained by secondary servers (which could be the first class servers for some other partitions of the naming data). Other replicated data are called *client caches*. This distinction makes the following scheme possible: replicas are responsible for the integrity of the UNS, read-only copies are responsible for offering clients consistent but possibly old data with higher availability, and caching is used to improve performance. The second class servers can also be used to reduce lookup overhead and improve service efficiency. A call-back mechanism is used by the first class servers to inform the second class servers about recent successful updates; this further reduces the number of queries to the first class servers. When the first class servers are not functioning, the second class servers can still answer enquiries. However no update can be made in this case, thus, no inconsistency can occur. Two types of queries to the first class service may be specified: one for "fast read" and the other for "slow read".

Finally, the problem of configuring the name service is addressed. Previous naming systems usually treat configuration in an ad hoc manner. The name service configuration involves two aspects: (a) The storage and maintenance of the replication configuration. For instance, the addition of a replica to or the removal of a replica from the existing replica set. (b) The storage and maintenance of the server configuration. For example, the movement or alternation of the responsibilities of servers. It is notable that previous work only studied these two aspects individually but not in combination. A distinguishing feature of the UNS is that both issues have been considered at the design stage and novel methods are used to allow dynamic service configuration to be done safely.

1.3 Plan of the Dissertation

The remainder of this dissertation is organised as follows. Chapter 2 describes the fundamentals of computer naming, including basic concepts, types of naming systems and principles. Chapter 3 considers the general aspects in the design of a universal name service. Chapter 4 presents a technique for dynamic construction of the UNS name spaces. Chapter 5 discusses maintenance of the naming database. Consistency mechanisms capable of supporting the UNS requirements are also investigated. Chapter 6 examines the conditions for correctness of dynamic name service reconfiguration and discusses how this can be archived. Chapter 7 presents the implementation of the UNS prototype, and finally Chapter 8 concludes the work and gives suggestions for further research.

Chapter 2

Computer Naming In General

2.1 Introduction to Naming

The general aspects of computer naming are reviewed in this chapter. It starts with some examples of naming, then followed by the conceptual definition of computer naming in Section 2.2. The particulars of distributed naming are explained in Section 2.3. Section 2.4 presents a brief survey of some computer name systems.

The first example of naming is from daily life. Suppose that there are two people Joe and Weili living in Britain and China respectively. Joe wants to send a letter to Weili; he writes the following on the envelope:

> Miss Lin, Weili
> P.O. Box 12345
> Beijing, 100083
> The People's Republic of China (PRC)

The mailing address contains a number of names: "Lin, Weili" for a person, "P.O. Box 12345" for an organisation, "Beijing" for a city, and "The People's Republic of China" for a country. After Joe posts the letter, one of the post offices in Britain will receive it and try to deliver it in some way. In fact, the mail may travel via several post offices before leaving the country. Later, it may be carried by a plane to China and classified, say at a central post office in Beijing, then sent to a regional post office where a postman will deliver the letter to Weili's pigeon-hole. No matter how complex the means to deliver the letter is, the mailing address plays an important role. Given the country name, the letter will go to China rather than to the US. A number of steps are involved to deliver

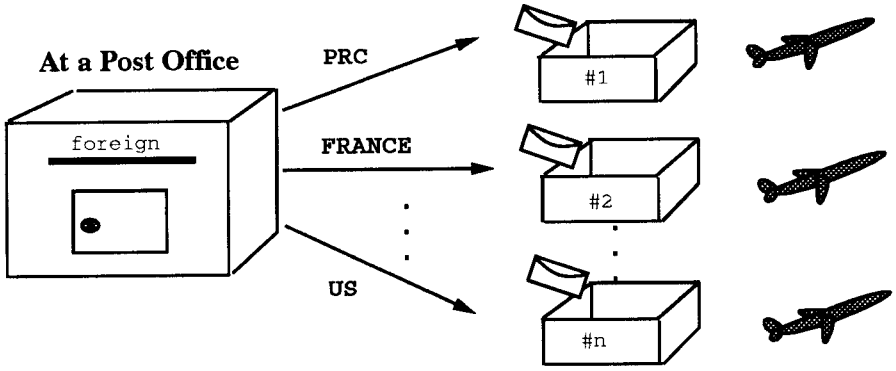


Figure 2.1: A Naming Example from Daily Life

the letter. Any post office concerned must have some information about the next possible step to take, and be able to contact the next post office in the chain. Figure 2.1 illustrates one possible step taken in a post office.

The second example, see Figure 2.2, is taken from the field of computing. A C program has an integer variable *sum*, and a symbol table generated by the compiler, which maps variable names to their memory addresses, e.g. from *sum* to *04B*, where the integer 15 is stored.

From these two examples, the following points are noted. Naming involves at least two entities, one holding a name by which the other is referred to. There are different kinds of names, some are meaningful to people, and others are not. Naming can be used for different purposes. Given an (source) entity that refers to another (target) entity by name, there is a mechanism which connects it with some information from which the target entity can be further identified. All these will be modelled in terms of computer naming in the next section.

2.2 General Concepts

A brief review of naming concepts in both centralised and distributed systems is presented in this section.

2.2.1 A Model for the Use of Names

A computer system typically involves a number of resources, such as files, processes, I/O units, memory segments, and so on. From an object-oriented point of view: a computer

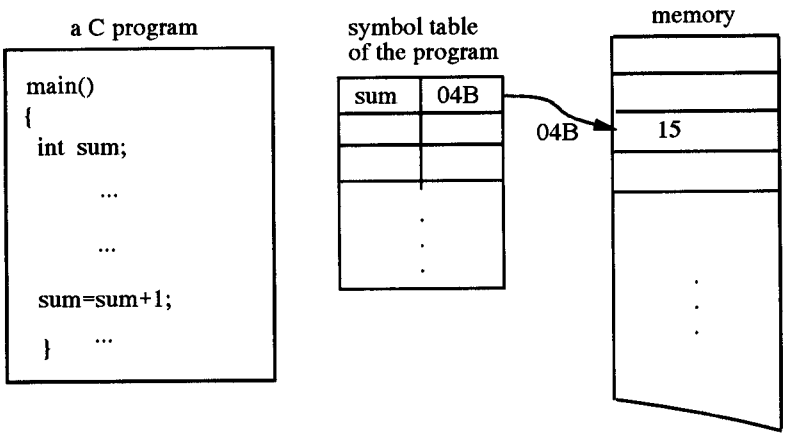


Figure 2.2: A Computer Naming Example

system can be seen as a manager of a variety of objects [Saltzer 79] (e.g. resources). To set up a **link**, i.e. a reference, between any two objects, a symbolic entity is used which identifies the link. [1] The symbolic entity is called the **name** or **identifier** for an object. For example, "Weili", "sum" and "04B" are names for a person, a variable and a memory cell respectively. It is possible for an object to have many names, or for the same name to be used by different objects for different purposes. In abstract terms, Saltzer defines a **context** of naming:

> *A context is a partial mapping from some names into some objects of the system. Arranging that a context shall map a name into an object is called* **binding** *that name to that object in that context.*

Figure 2.2 illustrates the source object (the program), the context (the symbol table), the binding of *sum* to *04B* and the target object *15*. There may be a *finite* number of lower level names and contexts [2] implementing the link between the source object and the target one. The target object must appear in one and only one context. The mechanism which connects the source object with its context is called a **closure**. For example, in Figure 2.2, the program interpreter implements the closure function by automatically using the symbol table as a context.

[1] A name is defined as a linguistic entity in [Linden 90] which singles out a particular entity from many. A name is also defined as a syntactic entity in [Comer 89]; name resolution is modelled as a string translation problem. To be generic, the word *symbolic* is used in this research.

[2] A name to target object binding usually involves a number of name mappings at several system levels. A lower level name or context is the one that a system gets after one or more name mappings. For example, given a variable name, a lower level name (an address of the memory) can be obtained by looking up the variable name in a program's symbolic table. See also **Name resolution**.

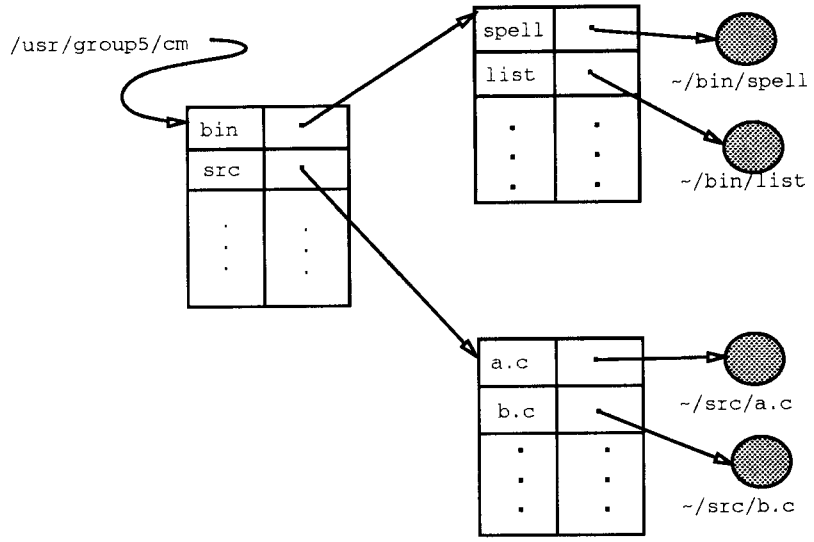


Figure 2.3: A Sample Naming Network

2.2.2 Basic Concepts

Naming Domain A naming domain contains all objects that can be named. For example: windows, files, file directories and memory segments of a computer system.

Naming Convention A naming convention defines the valid names to be accepted by a naming system and their interpretation, e.g. the UNIX file system defines a multi-component names which are composed of simple names such as "usr" and "bin", and separated by "/". For example, "/usr/bin/foo/bar" is a valid file name. Another naming convention found from the UNIX system is the generation and use of process identifiers (PIDs). A PID is an integer assigned to a process at its creation.

Name Space A name space contains all names acceptable by one and only one naming convention, e.g. all UNIX files form the UNIX file name space.

Name Network A name network can be defined as a directed graph [Comer 87], of which each vertex denotes a naming context and each edge denotes a reference to another context. A sample naming network is shown in Figure 2.3, where objects (files) are identified by multi-component names such as /usr/group5/cm/src/a.c. The context /usr/group5/cm contains other contexts, e.g. "bin" and "src". A file name can be resolved (see definition next) with the support of the naming network .

Name Resolution Name resolution defines a function which maps an input name to another name. There may be multiple steps of mapping before the lowest level name referring to the target object is returned. The lowest level name is used to

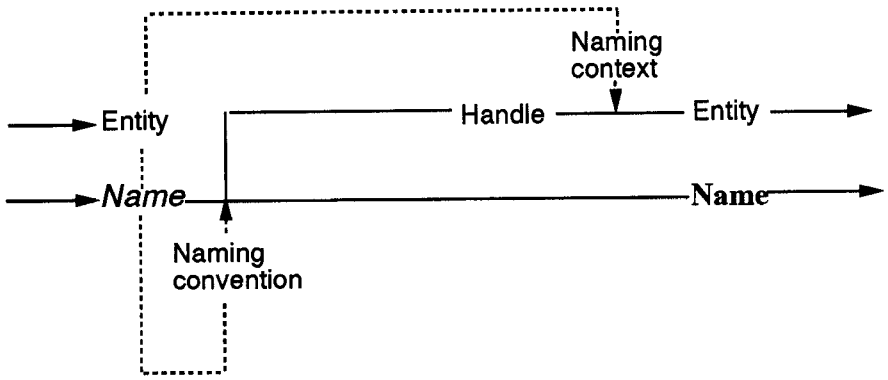


Figure 2.4: A Step in the General Name Resolution Model

locate the target object. A general name resolution model is defined in [Linden 90]. Figure 2.4 shows a single step that a name resolution involves. To resolve a name, a naming context derived from the entity and a naming convention are required to start with. The naming convention is used to derive a Handle and a **Name** (also called a remainder) from the given *Name*. By looking up the Handle from the naming context, a new entity is got. If the new entity is a naming context, the procedure continues, i.e. the remainder is used to extract a new handle and a new name, which is looked up in the new naming context, and so forth. Otherwise either the name being resolved is unbound, in that case an error is reported, or the name is resolved so that the procedure finishes.

2.3 Distributed Naming

The model and concepts defined in the last section apply to not only centralised computer systems but also distributed ones. Besides the basic concepts, some new ones must be introduced to solve naming problems encountered in distributed computing systems.

A distributed computing system is composed of a number of independent hardware and software components supported by one or more underlying computer networks. There is little doubt that more naming and binding exist in such a system than in a centralised one. Another interesting point is that these components are interdependent as well as autonomous. In describing interdependency, Lamport once defined a distributed system as a system in which the failure of a computer one has never heard of can make it impossible to get work done. On the other hand, these components are also autonomous: both in naming and management. Naming can be done independently within a name issuing authority, and naming information is maintained by each local authority rather than a central one.

Naming autonomy can be achieved by using hierarchical names. For instance, a naming domain can be partitioned either geographically or organisationally; the partitions are put into a tree structure. At each vertex on the tree, assignment of names can be done with no regard to that at other vertices. Hierarchy also makes unambiguous naming easy to do too. For example, the UNIX file system has hierarchical file names like "/sys/bin/man" and "/usr/group5/cm/bin/man", to refer to the two different print-out-the-manual programs.

New objects introduced by distributed systems include Host (Node), Socket, Gateway, Network, Service, Client etc. The introduction of new objects has enriched the types that a naming system has to support. On the other hand, the scale of the naming system increases as the distributed computing system extends. One of the main requirements of a distributed naming system is the ability to evolve the system for larger capacity and better functionality, i.e. the naming system can be extended as the distributed system grows.

In conclusion, a distributed naming system manages more objects than a centralised system. It is autonomous yet interdependent.

2.3.1 Federated Naming

Although there are different naming conventions in a centralised computer system, these conventions are used for different naming purposes and at different levels. There are few chances for them to be brought together to form a larger name space. However, in a distributed computing environment, communication and the sharing of resources become possible, which are also supported by the system. To enable the interconnections of various heterogeneous computer systems at all levels, it is valuable to make the naming systems work cooperatively. One approach to this is to set up federated naming systems [Schwartz 87, Peterson 88, Linden 90]. A **Federated Naming System** has the following features: [3]

- Disjoint naming domains before and after the federation.
- Naming contexts may be *exported* or *imported.* To export a naming context means to allow the context reachable from other naming systems. To import a naming context means to allow the context of another naming system to be reached by the current system. The imported or exported context does not have to be a separated context. Marking the exported or imported entities is required in order to distinguish them from the local entities.

[3] It is based on the notions in [Linden 90].

- The function, which makes a *foreign* context be reached by the naming network of the current system, does not affect the system's own naming convention.
- The federated naming space is flat.

In spite of the technical generality, the federated naming model lacks the support to the higher degree of global coherence and the performance provided by a centralised system. There are also problems in access transparency. For instance, the boundaries among the different naming domains are visible in a naming federation. The closure mechanism also becomes difficult to implement, especially for naming in heterogeneous computing systems.[4] As computer science and technology develop rapidly, global naming is in great demand. For instance, it is required that a new generation of file system have the ability of global naming [Birrell 91]. It is also required that the directory services or the electronic mail systems [X.500 88, X.400 84, Mockapetri88] be accessed globally in a coherent way. Furthermore, a federated naming system can not avoid scaling problem although efforts have been made to limit the size of a federation. However some of such systems may grow, and eventually become a global name service used by the whole world community (for example, the world telephone systems). The federated naming model may cope with a distributed system with a moderate size, stronger autonomy, and restrict sharing, but not with very large distributed systems.

2.3.2 Global Naming

A **Global Naming System** distinguishes itself from a federated one. It is defined as:

- A logically centralised but physically distributed naming system
- with globally unique and accessible names and
- a uniform interface for the client.

Any participant of the system must adhere to the agreements for accessing the system such as the name service access protocols and the naming convention. A centralised naming system differs from a distributed naming system in the way it is constructed. For instance, it will completely break down when the underlying computer system fails. A common model to implement a distributed system is the Client/Server model [Birrell 82, Needham 82, Coulouris 88]. A **Name Server** provides operations for

[4] Problems of this kind are addressed in [Schwartz 87].

manipulating names. In particular, it maps names for objects to a set of attributes (properties), such as mailboxes, network addresses, and machine names. An **Attribute** is also named and assigned with a number of values. For example, the value of a network address is 128.232.0.10. A **Name Service** is composed of one or more **Name Servers** which cooperate to resolve names.

2.4 A Brief Survey of Naming at Different Levels

This survey concentrates on Different Types of distributed naming systems, each of which may be considered as a component of a general naming model for distributed computing systems, which is defined in Section 2.4.1.

2.4.1 An Open Naming System Model

A typical pattern of naming objects is given in Figure 2.5-(a), where a client locates the object in steps including descriptive naming (for example, using the Profile Name System [Peterson 88]), primitive naming (for example, using Grapevine [Birrell 82]) before the object manager is reached. Alternatively, a client uses the register service (name service, see [Birrell 82]) and the object manager only to locate the object. It is also possible for the client to use the object manager only.[5] Besides the difference in function, descriptive naming is concerned with intelligent naming, while primitive naming and integrated naming (with the object manager) emphasizes unambiguity and efficiency.

A model for an open naming system is described by Figure 2.5-(b). In contrast to a closed naming system, the open system is composed of a number of optional components, e.g. A, B, C and D. It is not necessary for a naming system to have every component. Likewise, some of the components may be bypassed by the client. For instance, a file system does not usually need attribute-based naming. With the primitive name for an object obtained from a previous query to a descriptive name service in hand, a user may not bother to obtain it again on a later access to the same object.

[5]The Stanford approach [Cheriton 88] to a decentralised naming system focuses on naming handled directly by the object manager. Client-name caching and multicast are exploited to implement the mapping of names for good performance and fault tolerance.

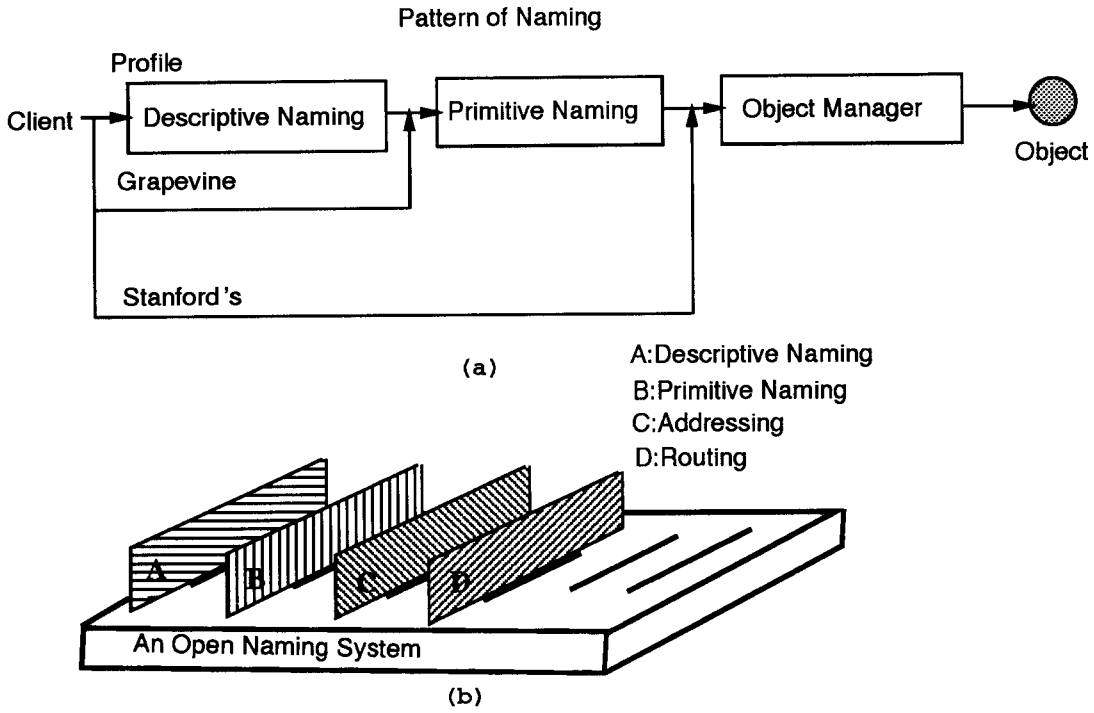


Figure 2.5: Naming Pattern and Model

2.4.2 Profile

Profile is an attribute-based (descriptive) naming service for a large internet [Peterson 88]. It has three main components: a confederation of attribute-based name servers, a name space abstraction that unifies the name servers, and a user interface that integrates the name space with existing naming systems. There are multiple naming authorities, i.e. name servers, responsible for different set of principles, which are users or organisations that sponsor some collections of resources. The name servers form a loosely coupled confederation with a non-hierarchical name space. Profile supports a name space abstraction that unifies the confederation by providing an extended syntax for specifying attribute-based names, and a discipline for contacting a multiplicity of name servers etc.

2.4.3 Grapevine

The Grapevine system [Birrell 82] developed at the Xerox PARC provided facilities for mail delivery, naming, authentication and locating. The message server, the file server as well as user programs were all clients of the registration server - a primitive naming service. Clients view a single Grapevine server although it was implemented by a multicomputer system. A single naming convention was used. In spite of under the control of a single

authority, the system was running on the Xerox research internet which connected sites not only in the United States, but also in England, Canada and France.

2.4.4 Internet Addressing and Routing

Naming on the higher levels of distributed systems has been given in the previous sections. In this subsection, examples of naming at the lower levels, i.e. addressing and routing on the Internet, are illustrated. One should not be confused by the term *address*; it is in fact the name to which an object is bound.

Addressing

The Internet uses 32-bit binary addresses as universal machine identifiers which are composed of three parts: *classid*, *netid* and *hostid*. They are three classes of address; each allows certain range of netids and hostids. For example, class A address allows a few hundred networks with over a million hosts each. The ARPANET has the class A address 10.0.0.0.. Example connections of the Internet by Comer [Comer 88] are shown in Figure 2.6. The Internet address refers to network connections, thus hosts with more than one connections have multiple addresses, such that host MERLIN has two: 128.10.2.2 and 192.5.48.3. The Internet address can be used not only by networks, hosts, but also all hosts on a network (broadcasting).

Routing

Routing is the process to decide the path for sending packets given the destination address. IP routing concerns where to send a datagram based on its destination address. If the destination host lies on the same address to which the source host is connected, the route is direct, otherwise it is indirect, and the datagram must be sent to a gateway for delivery. Sending a datagram over a network involves resolving its address to the physical address, encapsulating the datagram, and sending the frame using the underlying hardware.

2.4.5 Summary

In this chapter, naming concepts, models and techniques have been introduced, with emphasis on naming in distributed systems. A notable feature of distributed naming is the expansion of the system in both number of objects and types. Some existing naming systems are outlined for their distinguishing identities: namely, descriptive naming, primitive naming and lower level naming. An open naming system architecture is defined to accommodate these different types of naming.

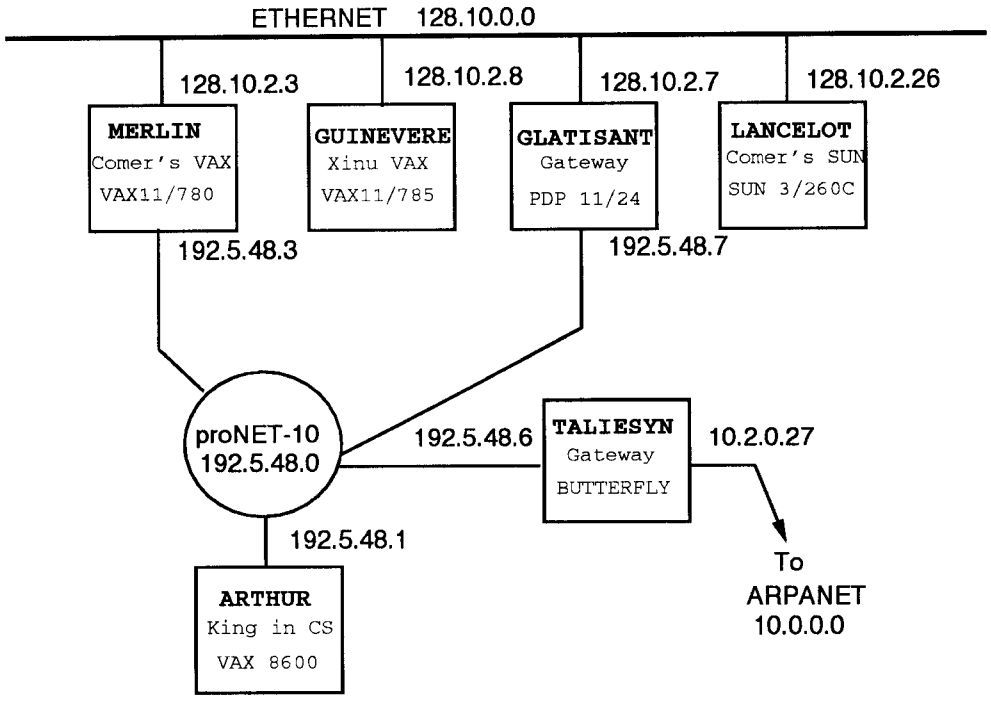


Figure 2.6: Example Connections for an Ethernet, Ring Network and ARPANET

Chapter 3

Designing a Universal Name Service

3.1 Introduction

Although constructing distributed systems appears to be more difficult than people foresaw, a great many distributed systems, such as distributed computing systems, mail systems, language systems, storage systems and name systems, have been studied and implemented [Needham 82, Birrell 82, Walker 83, Bacon 87, Mullender 87, Liskov 88]. Despite implementational difficulty, distributed systems have shown great potential because of their availability, reliability, fault tolerance, as well as scalability. In this chapter, some general aspects of designing a very large distributed naming system - the Universal Name Service (UNS), are discussed, based on the experience of previous work mentioned above. This section motivates the design and explains the goals of the UNS. Section 3.2 gives a brief survey of related work. Section 3.3 describes the design of UNS in detail, and section 3.4 presents the abstract interface to the UNS.

3.1.1 Motivation

Historically, name systems have been implemented initially in a centralised manner, such as the ARPANET name server maintained by SRI-NIC. As distributed computing systems develop, more and more distributed or decentralised name systems have emerged, either in real life or in designs and prototypes, notably Grapevine [Birrell 82], Clearinghouse [Oppen 83], the Global Name Service (GNS) [Lampson 86], the DARPA Domain Name System [Mockapetri88] and QUIPU [Kille 89]. The CCITT X.500 / ISO 9594 Directory

Service [X.500 88] has been adopted as an international standard. A desirable name service for future wide area computer networks should be distributed, accessible from anywhere in the world, reliable, inter-operable among heterogeneous networks and computer systems, secure and fault-tolerant. Growth of such a system in order to meet increasing demands should ideally be carried out in small steps without changes of major techniques. Although many naming systems or designs exist, few have adequate abilities to tackle scalability, availability and fault tolerance. The scalability problem has been discussed in the previous chapter. Furthermore, it is observed that although using stale naming data (hints) at the application level is acceptable in most cases, as long as it is detectable and recoverable, stronger naming data integrity should be maintained to provide a better guarantee of finding objects, especially when a high degree of availability is required. Configuring a name service is usually done in an ad hoc manner, leading to unexpected interruptions or a great deal of human intervention when the system is reconfigured. It is necessary to further study distributed naming issues in order to build a more efficient and robust global name service.

In this chapter, the design of a universal name service is presented, and its features are described.

3.1.2 Design Goals

- Flexibility of name space restructuring
  The UNS name space should be able to grow to accommodate other name spaces. A directory/name server should also be allowed to move from one server/address to another.

- Dynamic service configuration
  Dynamic service configuration deals with, for instance, changing the replication set or moving servers around when the UNS is operating. Work should be done on setting conditions and developing algorithms to make such configuration transparent to clients.

- Fault tolerance
  The UNS should continue to operate in the presence of network failures, site failures or client failures. Partial failures of the service should not cause the entire naming system to become unavailable. The effect of failures should preferably be localised.

- High availability
  The UNS is a frequently used service and is used by many other services. For

instance, a message delivery server may cooperate with a nearest UNS server for good mail service. A user can send a query to a UNS server connected to the same local network for quick response.

- Strong data integrity
  Most name services provide high availability for both queries and updates, but few of them offer accurate naming data. The UNS attempts to enable its clients to find out the most recent information by maintaining strong integrity of naming data. There is a trade-off between data consistency and update availability. Prototyping with the UNS will show later that under naming circumstances, high availability can be achieved without sacrificing data integrity.

3.2 Related Work

A survey of naming systems of different types has been given in Chapter 2. This section presents a brief survey of some global naming systems.

3.2.1 The Global Name Service (GNS)

The GNS designed by Lampson et al. [Lampson 86] is a basis for resource location, mail addressing and authentication. Issues such as high availability, large size, continuing evolution, fault isolation and tolerance of mistrust are addressed. The GNS, however, lacks the ability to resolve unique Directory Identifiers (DIs) as the system grows, which restricts its ability to restructure name spaces in case of need. The GNS uses the *sweep* algorithm to propagate updates, which provides no guarantee that clients will discover the most recent data. Dynamic service configuration is not fully supported.

3.2.2 The Domain Name System

The Domain Name System (DNS) [Mockapetri88] is a well known and widely used name service. A typical domain name has the following form: XX.LCS.MIT.EDU. There is no implementation of unique identifiers such as in the GNS and the UNS. The system is neither migration nor naming transparent. For example, if one wants to add US as a new root to the current system, it will cause a large number of domain names to be changed, i.e. XX.LCS.MIT.EDU becomes XX.LCS.MIT.EDU.US. This is impractical. Although a domain name looks like a UNS Name, it is relative to a context which may be changed, while a global unique identifier is not.

3.3 Overview of the Universal Name Service

This section discusses general aspects concerning the design of the UNS. The UNS is a primitive naming system addressing issues such as large scale, evolution, reconstruction, fault tolerance, availability and reliability.

3.3.1 The UNS Fundamentals

UNS Naming Domain : Possible **objects** in the UNS naming domain are end users, directories, indexes, mailboxes, machines, services, and other resources. A **directory** is an entity which contains a naming context. Each object is represented in a directory by a single, named **entry**. Each entry is composed of a number of properties, represented by a set of **attributes**. An attribute has a **type** which defines the generic class of the property, and a **value** which is a specific instance of the type. In terms of the ASN.1 [ISO86], an attribute, an entry and a directory can be specified as follows.

```
Attribute ::= SEQUENCE {
        type        AttributeType,
        values      SET OF AttributeValue }
Entry ::= SET {
        name        [0] PrintableString,
        attributes  [1] SET OF Attribute }
Directory ::= SET {
        version     [0] TimeStamp,
        entry       [1] SEQUENCE OF Entry }
Index ::= SET {
        version     [0] TimeStamp,
        entry       [1] SEQUENCE OF Entry }
```

Example attributes are:

```
internet address  = 192.5.48.1
password          = xx#ztq0ô
user id           = cm119
```

An example entry is:

```
Chaoying Ma ⟶ {
        password        = NI09U8cmCV9d2
        nfs server      = { ely, cormorant }
```

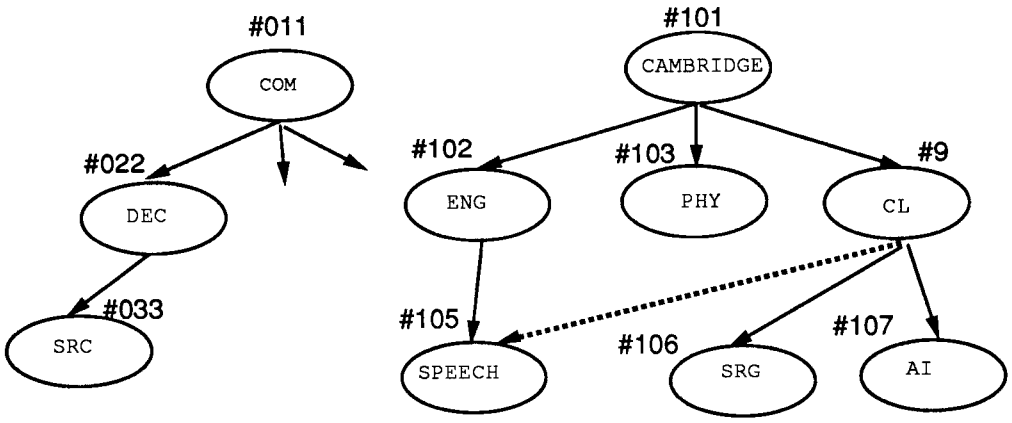


Figure 3.1: Examples of the UNS Name Spaces

user id	= cm
e-mail address	= cm@uk.ac.cam.cl}

Any object is represented by an entry consisting of a name and a set of attributes. An example object class may be as follows:
Object-Class = { index, directory, individual, group }

UNS Name Space : A UNS Name space contains all valid names within an administration. If a name space is contained by another name space, the name space is called a sub-name-space. For instance, suppose CL is the name space for the Computer Laboratory at Cambridge, and CAM is that for Cambridge University, then CL is a sub-name-space of CAM. A tree structure is frequently used in name server designs, despite the fact that it allows each of its nodes to have only one name. Alternatively, a rooted, acyclic directed graph can avoid the uniqueness constraint of trees. Furthermore, a group of such directed graphs allows several name spaces to coexist, forming the UNS name space. Figure 3.1 illustrates some example sub-name-spaces of the UNS. Note that each directory is associated with a "UDI" besides its relative name. For instance, the directory with a relative name "DEC" is associated with "#022". In the next chapter, it will be seen that UDIs are used to locate directories whose positions in the name space may change.

Name Structure : The name structures allowed by the UNS naming convention are defined by the following Backus-Naur Form:

simple name ::= <any printable string without />
relative name::= <simple name> | <relative name> / <simple name>

DN ::= UDI | UDI / <relative name>
IN ::= < relative name>

- **UDI**: A UDI is a global Unique Directory Identifier. The generation of a UDI requires an inter-domain agreement that is adhered to by all parties participating in the UNS. Methods of producing such identifiers independently [1] are beyond the scope of the thesis. The reason why each directory should be given a UDI will be explained in Chapter 4.
- **Absolute Name**: An absolute name has the form (DN,IN), which consists of two parts: where DN denotes a distinguished Directory Name, and IN denotes an Individual Name, which is relative to the context specified by DN. A distinguished Directory Name (DN) can be either a UDI or a string of characters starting with a UDI, for example, "#123/cam/cl". An IN can be a path name which designates a node in a value tree, or any form that a local administration defines; for example, "srg/cm".
- **Soft Link**: The value of a name is a soft-link, which is another DN. An example is given in Figure 3.1, where the value of #9/SPEECH is the soft-link #101/ENG/SPEECH. [2]

3.3.2 Name Resolution

Under the assumption that all names are structured and only context relative naming is allowed, [3] the UNS name resolution can be defined as follows:

Rule1: Looking up DN = UDI/$name_1/name_2/.../name_n$ yields the directory d_1 where $name_1$ is defined, looking up d_1 yields the directory d_2 where $name_2$ is defined, and so forth. The process ends when no more relative names, i.e. $name_i$, $1 \leq i \leq n$, are left unresolved.
Rule2: The name IN is defined in the directory named by DN. The rule for resolving IN is the same as **Rule1**.
Rule3: If looking up a name UDI$_1/n_1/n_2/.../n_m$ yields a soft-link UDI$_2/n'_1/n'_2/.../n'_k$, **Rule1** applies to resolve the soft-link.

To start a name resolution with a given name, a context is needed. A starting context

[1] There are many methods to generate unique identifiers independently. For instance, a UDI may be formed by combining a wall clock reading and a site id.

[2] See also Chapter 4.

[3] An example for non-context-relative naming can be found in [Mann 87].

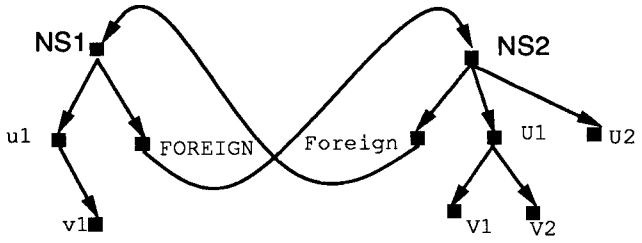


Figure 3.2: Mutually Encapsulated Name Spaces

can be either a global one or a local one. In the UNS design, a name resolution normally starts from a global context as specified by the DN at the first time the name is looked up. Caching may be used to avoid going to the same context again when the name is encountered later. For example, if the name (#101/CL/SRG, xyz) is given, see Figure 3.1, the directory CAMBRIDGE is the starting context. In order to resolve (#101/CL/SRG, xyz), the directory CAMBRIDGE is first looked up to get CL, then the directory CL is looked up to get SRG, and the directory SRG is finally looked up to resolve "xyz". In the UNS, resolution of DNs is implemented differently from that of INs. The distinction is made because their naming semantics is different: INs can belong to heterogeneous naming architectures, but DNs cannot. In a local environment, for instance, within the Engineering Department, the starting context for name SPEECH/m.phil/xyz is directory SPEECH.

Once a name resolution starts, it must be terminated in a finite number of steps. In general, when name resolution yields a *real* value rather than a context name, the procedure ends. A naming network must be configured very carefully to avoid loops. Hierarchy structures and directed acyclic graphs are cycle free. A well formed acyclic naming network may become a part of a cyclic naming network, however, which for example, serves a federated naming system. Figure 3.2 shows two mutually encapsulated name spaces, which will lead to non-terminating name resolution. In the UNS, only rooted, directed acyclic graphs are permitted to form name spaces.

3.3.3 Name Service and its Components

A name service assigns names to objects [4] and resolves given names. A name service consists of six major components: a naming context, a naming network, a naming convention, a name resolution mechanism, a naming database and a name server. As described

[4] Note that a context is also an object.

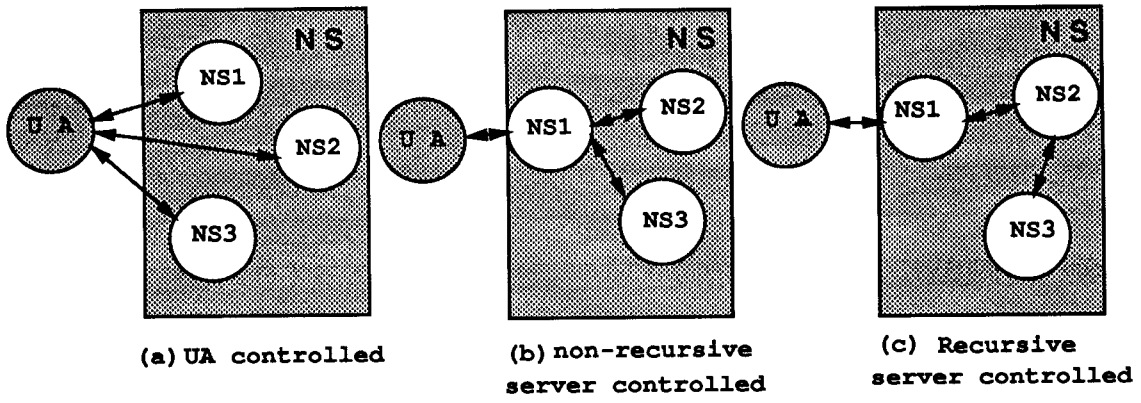


Figure 3.3: Three Kinds of Navigation

above, names are constructed according to naming conventions. A name service applies a name resolution mechanism to a set of naming contexts in order to resolve names. Naming contexts are organised as a naming network, in which contexts are stored by servers. Distributing contexts among the servers reflects the configuration of a name service. In a distributed environment, a number of name servers work collectively to provide a name service. A name database is partitioned and distributed among servers. A name server can be viewed as an instance of a name service. Note that different service instances may or may not contain the same naming contexts; the former reflects replication, while the latter reflects partitioning of the name space. Replication and partitioning can be done in many ways; these will be addressed in Chapter 5.

In many naming system designs, a User Agent (UA) component is used. A UA is a collection of programs which provide the interface to name servers and locate servers at the start of name resolution.

Navigation is the process of forwarding an operation to a name server for execution. Once a naming network is set up, navigation can be done step by step following pointers which refer to other servers (see Figure 2.3). Since a server may store more than one context, the number of navigation steps required should be less than the number of closure steps. As shown in Figure 3.3, navigation can be controlled by a UA or a server. In the former, the UA is responsible for communication with one or more servers to resolve a given name. In the latter, the server assumes overall responsibility for contacting other servers in a non-recursive or recursive way. Figure 3.3 illustrates the three ways in which a navigation can be done. In a very large distributed environment, using (a) or (b) only may be impossible because of access restrictions. If there is underlying broadcast or multicast support, (a)

and (b) are more efficient than (c), but (c) is generic. The UNS combines all three into a hybrid navigation style. For instance, a UA may contact some local servers concurrently using (a) when the name resolution starts, then one of the servers being contacted fulfils the rest of the task using (c).

Another important issue on designing the UNS is **security**. There are two aspects involved: authentication and protection. Authentication is concerned with identification issues, such as setting up a secure channel between two servers, or between a server and a client. Protection is concerned with two issues: (a) a server should provide services only to those clients who are authorised to use it, and (b) only an authorised server should offer the required service. Security naming systems can be provided by appropriate access-control mechanisms. For instance, an access control list can be attached to each object entry which needs to be protected. Each time that the entry is referred to by a principal, the access rights of the principal are checked. The access right may be any combination of {read, update, test}. This research does not focus on security issues, and the design does not make any significant difficulty for implementing security mechanisms.

3.4 An Abstract Interface for the UNS

The previous sections have described the abstract structure of the UNS in terms of name spaces, entries and attributes. This section presents the abstract interface which defines the access protocols for the UNS. The abstract operations are specified in terms of ASN.1 [ISO86]. The protocols are layered and will be supported by an RPC (Remote Procedure Call) mechanism.

3.4.1 The UNS Architecture

The client accesses the UNS via the User Agent (UA) which is composed of a number of abstract operations for querying, updating or manipulating the naming data. The UA resides on every machine where the client will make use of the UNS. As shown in Figure 3.4, the client accesses the UNS through the UNS client interface; there is also a UNS administration interface which provides the UNS system administrator with a tool for managing the UNS database. The next two sections describe the abstract UNS client and administration interface respectively.

3.4.2 Client Interface

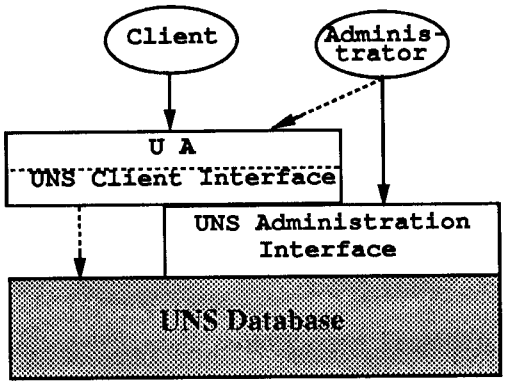


Figure 3.4: The Abstract Architecture of the UNS

	Operations for Entries
EnumerateEntry	Enumerate all the attributes associated with a named entry
AddEntry	Create then add an entry to a specified directory
DeleteEntry	Delete the named entry from the directory
ReadEntry	Read the attributes of the named entry
ModifyEntry	Modify the entry in a directory
TestEntry	Test whether some attributes are there
	Operations for Directories
ListDirectory	List all the directory's child entries
CreateDirectory	Create a directory entity
RemoveDirectory	Remove a directory entity
MoveDirectory	Move a child directory from the current parent directory to another one
	Others
CheckGoodness	Check the version number of a directory

Figure 3.5: Summary of UNS Client Interface

First of all, the client interface is pictured in Figure 3.5. The following error signals can be raised by the operations:

1. InputError: This includes error signals such as InvalidName, InvalidAttributeType, etc.
2. AccessError: This contains all access violation errors such as NoRight, NoUpdateRight etc.
3. ReadError: This could be any of the following: UnknownEntry, UndefinedAttribute etc.
4. GeneralError: This describes general errors such as ServerNotAvailable, ClientTimeOut, ServerTimeOut etc.
5. TestError: This contains errors such as TypeNotMatch, ValueNotMatch etc.

Type checking and access control are done whenever an operation is invoked.

The following operations are defined:
ReadEntry: The ReadEntry operation requests read access to an entry, and returns the specified attributes in (type, value) pairs. If the Accurate option is chosen, the server provides the most recent data, otherwise the server responds as quickly as possible with data which may well be out of date. When a failure occurs, a general error is signalled.

```
ReadEntry ::= ABSTRACT-OPERATION
        INPUT       ReadInput
        RESULT      ReadResult
        ERRORS      { InputError,ReadError,AccessError}
ReadInput ::= SET {
        entry       [0] Name,
        types       [1] AttributeType,
        option      [2] {Accurate, Hint }}
ReadResult ::= SET {
        attributes  [0] Attribute
                    [1] SET OF ReadError }
```

EnumerateEntry: The EnumerateEntry operation checks both the INPUT and the access rights. If no error condition is received, it returns all the attributes associated with the named entry. Similarly to the ReadEntry operation, the Accurate or Hint option can be chosen by the client, and failures, if any, are reported.

```
EnumerateEntry ::= ABSTRACT-OPERATION
        INPUT       EnumerateInput
        RESULT      EnumerateResult
        ERRORS      { InputError, AccessError}
EnumerateInput ::= SET {
```

```
        entry           [0] Name,
        option          [1] { Accurate,Hint }}
EnumerateResult ::= SET {
        attributes      [0] Attribute
                        [1] SET OF ReadError }
```

ModifyEntry: The ModifyEntry operation requests update access to the named entry. It can add or remove attributes, or replace an old attribute value with a new one. The modification can be a single operation or a group of operations. In the latter case, a transaction may be used if it is requested. This is not usually supported by a name service however, due to its cost. As indicated in previous sections, an AttributeType may be a soft-link, whose value is a DN. The ModifyEntry operation can be used to add or remove the soft-link for an entry.

```
ModifyEntry ::= ABSTRACT-OPERATION
        INPUT           ModifyInput
        RESULT          ModifyResult
        ERRORS          { InputError,ModifyError,AccessError}
ModifyInput ::= SET {
        entry           [0] Name,
        modifications   [1] SET OF Modification }
Modification ::= SEQUENCE {
        type AttributeType,
        CHOICE {
                        add    [0] AttributeValue,
                        delete [1] AttributeValue,
                        replace[2] SEQUENCE {
                                   old AttributeValue,
                                   new AttributeValue }}}
ModifyResult ::= NULL
```

AddEntry: The AddEntry operation creates the entry for an object to be named. The entry name is therefore the object name. There may be zero or more attributes associated with the entry. The created entry is then added to the specified directory.

```
AddEntry ::= ABSTRACT-OPERATION
        INPUT           AddInput
        RESULT          AddResult
        ERRORS          { InputError,AddError,AccessError }
AddInput ::= SET {
        directory       [0] Name,
        entry           [1] Name,
        attributes      [2] SET OF Attribute }
AddResult ::= NULL
```

TestEntry: the operation compares the specified attributes with those stored by a directory. If they do not match then a TestError condition will be returned.

```
TestEntry ::= ABSTRACT-OPERATION
        INPUT           TestInput
        RESULT          TestResult
        ERRORS          { InputError,TestError,AccessError }
TestInput ::= SET {
        entry           [0] Name,
        attributes      [2] SET OF Attribute }
TestResult ::= NULL
```

DeleteEntry: The DeleteEntry operation removes the named entry and its associated attributes.

```
DeleteEntry ::= ABSTRACT-OPERATION
        INPUT       DeleteInput
        RESULT      DeleteResult
        ERRORS      { InputError,DeleteError,AccessError}
DeleteInput ::= Entry
DeleteResult ::= NULL
```

ListDirectory: The ListDirectory operation returns all entry names of its child directories. Wildcard symbols may be used to select a subset of entry names.

```
ListDirectory ::= ABSTRACT-OPERATION
        INPUT    ListInput
        RESULT   ListResult
        ERRORS   { InputError,ListError,AccessError}
ListInput ::= SET {
        directory  [0] Name
        option     [1] {Accurate,Hint} }
ListResult ::= SET OF Name
```

CreateDirectory: The CreateDirectory operation sets up an empty directory object with the default attributes, e.g. an access control list. The CreateEntry operation can then be used to add object entries to the newly created directory. If a server name is given, a check is made to find out whether the directory can be kept by the server or not. If not, a NotHere error message is signalled. If no server name is supplied, the system will choose a suitable server for the directory. A Unique Directory Identifier is returned if the operation succeeds. The operation also causes the corresponding parent directory to be modified.

```
CreateDirectory ::= ABSTRACT-OPERATION
        INPUT    CreateInput
        RESULT   CreateResult
        ERRORS   { InputError,CreateError,AccessError}
CreateInput ::= SET {
        directory  [0] Name,
        server     [1] Name OPTIONAL }
CreateResult ::= SET {
        udi        UID }
```

RemoveDirectory: The operation destroys the specified directory. If the directory is replicated, the operation removes every copy of it whether or not a server name is supplied.

```
RemoveDirectory ::= ABSTRACT-OPERATION
        INPUT    RemoveInput
        RESULT   RemoveResult
        ERRORS   { InputError,RemoveError,AccessError}
RemoveInput ::= SET {
        directory  [0] Name,
        server     [1] Name OPTIONAL }
RemoveResult ::= NULL
```

MoveDirectory: The operation changes the parent directory of the specified child directory. There is no need to indicate the old parent since it can be deduced from the name of the child directory. Section 4.3 explains this in detail.

```
MoveDirectory ::= ABSTRACT-OPERATION
        INPUT           MoveInput
        RESULT          MoveResult
        ERRORS          { InputError,MoveError,AccessError}
MoveInput ::= SET {
        child-directory   [0] Name,
        parent-directory  [1] Name }
MoveResult ::= NULL
```

CheckGoodness: When a directory is created, a time-to-live stamp may be associated with it. This operation verifies the time-stamp of a directory to check whether it has expired or not. The operation also provides the client with a means to find out if a directory is still there.

```
CheckGoodness ::= ABSTRACT-OPERATION
        INPUT    CheckInput
        RESULT   CheckResult
        ERRORS   { InputError,CheckError,AccessError}
CheckInput ::= Directory
CheckResult ::= Timestamp
```

3.4.3 Administration Interface

The operations described in the administration interface are used to manage the name spaces, the name servers , the index (see the next chapter for more details) and the directory copies. These operations are normally carried out by system or network managers. A summary of the interface is given in Figure 3.6. The operations are:

SetIndex: This operation creates the index object to resolve the GUDIs. A well known server is specified to store the index. GUDIs, as well as server identifiers (SIDs), are added by the InsertUDI operation. If the server is authorised to do updates, it is an authorised server, otherwise it is a server that simply stores a copy but has no right to change the copy.

```
SetIndex ::= ABSTRACT-OPERATION
        INPUT    SetInput
        RESULT   SetResult
        ERRORS   { InputError,SetError,AccessError}
SetInput ::= SET {
        server      [0] Name,
        wellknown   [1] NetworkAddress,
        option      [2] { AuthorisedCopy,Copy }OPTIONAL }
SetResult ::= UID
```

DestroyIndex: This operation removes the index and all its authorised copies. It is only used when a catastrophe occurs on many authorised servers or their connection networks.

	Operations for Indexes
SetIndex	Create a new index
DestroyIndex	Destroy an index
PrintIndex	Print out all the content of an index
AbandonUDI	Throw away the UDI
ModifyUDI	Modify the UDI mapping
InsertUDI	Insert the UDI to an index
	Operations for Replicas
AddReplica	Add a new replica to a specified server
RemoveReplica	Remove a replica from a specified server
GetRSet	Get the replica set
	Operations for Servers
SetUpServer	Set up a new server
DisableServer	Disable a server
EnableServer	Enable a server
RecoverServer	Restore the current state of the server
ReadState	Get snapshot of the server state
	Operations for Namespaces
NewNamespace	Create a new name space
DestroyNamespace	Destroy an existing name space
SetDefaultNamespace	Set the default namespace for the server

Figure 3.6: Summary of UNS Administration Interface

```
DestroyIndex ::= ABSTRACT-OPERATION
        INPUT     DestroyInput
        RESULT    DestroyResult
        ERRORS    { InputError,DestroyError,AccessError}
DestroyInput ::= SET {
        index-id    [0] UID }
DestoryResult ::= NULL
```

PrintIndex: This operation provides a current snapshot of the index for management purposes.

```
PrintIndex ::= ABSTRACT-OPERATION
        INPUT     PrintInput
        RESULT    PrintResult
        ERRORS    { InputError,PrintError,AccessError}
PrintInput ::= SET {
        index-id    [0] UID }
PrintResult ::= SEQUENCE {
        GUDI      UID,
        CHOICE {
                  GUDI [0] UID,
                  address[1] NetworkAddress}}
```

AbandonUDI: This operation disables and invalidates the given UDI.

```
AbandonUDI ::= ABSTRACT-OPERATION
        INPUT     AbandonInput
        RESULT    AbandonResult
        ERRORS    { InputError,AbandonError,AccessError}
AbandonInput ::= SET {
        GUDI      UID }
AbandonResult ::= NULL
```

ModifyUDI: This operation requests update access to the index. It adds server identifiers (SIDs) or addresses to, or removes SIDs or server addresses from the value set of the specified GUDI. It may replace an existing SID or address with a new one. Since it concerns the UNS configuration, only one operation is allowed at a time (see the safety conditions given in Chapter 6).

```
ModifyUDI ::= ABSTRACT-OPERATION
        INPUT     ModifyInput
        RESULT    ModifyResult
        ERRORS    { InputError,ModifyError,AccessError}
ModifyInput ::= SET {
        GUDI      UID,
        CHOICE {
                  add       [0] CHOICE {UID,NetworkAddress}
                  delete    [1] CHOICE {UID,NetworkAddress}
                  replace   [2] CHOICE {UID,NetworkAddress} }
ModifyResult ::= NULL
```

InsertUDI: This operation inserts a GUDI with the indicated values to the index.

```
InsertUDI ::= ABSTRACT-OPERATION
```

```
        INPUT     InsertInput
        RESULT    InsertResult
        ERRORS    { InputError,InsertError,AccessError}
InsertInput ::= SET {
        GUDI      [0] UID,
        value     [1] CHOICE {UID,NetworkAddress} }
InsertResult ::= NULL
```

The following operations deal with service configuration:

AddReplica: This operation adds a copy of the named directory to the indicated server. The replica set of the directory is changed.

```
AddReplica ::= ABSTRACT-OPERATION
        INPUT     AddInput
        RESULT    AddResult
        ERRORS    { InputError,InsertError,AccessError}
AddInput ::= SET {
        server    [0] Name,
        directory [1] Name }
AddResult ::= NULL
```

RemoveReplica: This operation removes the named replica from the specified server. The replica set of the directory is changed.

```
RemoveReplica ::= ABSTRACT-OPERATION
        INPUT     RemoveInput
        RESULT    RemoveResult
        ERRORS    { InputError,InsertError,AccessError}
RemoveInput ::= SET {
        server    [0] Name,
        directory [1] Name }
RemoveResult ::= NULL
```

GetNset: This operation returns the current members of the server name set of the directory if "Accurate" is indicated, otherwise anything available.

```
GetNset ::= ABSTRACT-OPERATION
        INPUT     GetInput
        RESULT    GetResult
        ERRORS    { InputError,GetError,AccessError}
GetInput ::= SET {
        directory [0] Name,
        option    [1] {Accurate, Hint}}
GetResult ::= SET OF Name
```

SetUpServer: This operation initiates a new UNS server with default configuration. The CreateDirectory or AddReplica operation will add the naming context to the server.

```
SetUpServer ::= ABSTRACT-OPERATION
        INPUT     SetupInput
        RESULT    SetupResult
        ERRORS    { InputError,SetupError,AccessError}
SetupInput ::= SET {
        server    [0] Name,
        attribute [1] SET OF Attributes }
SetupResult ::= NULL
```

DisableServer: This operation changes the mode of the server from Working to Suspending when the server is unable to continue its service.

```
DisableServer ::= ABSTRACT-OPERATION
        INPUT     DisableInput
        RESULT    DisableResult
        ERRORS    { InputError,DisableError,AccessError}
DisableInput ::= SET {
        server    [0] Name,
        mode      [1] Suspending }
DisableResult ::= NULL
```

EnableServer: This operation changes the mode of the server from Suspending to Working when the server is restored.

```
EnableServer ::= ABSTRACT-OPERATION
        INPUT     EnableInput
        RESULT    EnableResult
        ERRORS    { InputError,EnableError,AccessError}
EnableInput ::= SET {
        server    [0] Name,
        mode      [1] Working }
EnableResult ::= NULL
```

RecoverServer: This operation causes the server to enter the recovery mode to restore its corrupted naming contexts.

```
RecoverServer ::= ABSTRACT-OPERATION
        INPUT     RecoverInput
        RESULT    RecoverResult
        ERRORS    { InputError,RecoverError,AccessError}
RecoverInput ::= SET {
        server    [0] Name,
        directory [1] Name,
        mode      [2] Recover }
RecoverResult ::= NULL
```

ReadState: This operation provides a snapshot of the server state.

```
ReadState ::= ABSTRACT-OPERATION
        INPUT     ReadInput
        RESULT    ReadResult
        ERRORS    { InputError,ReadError,AccessError}
ReadInput ::= SET {
        server    [0] Name }
ReadResult ::= SET OF Attributes
```

NewNamespace: This operation sets up the root directory for a new name space.

```
NewNamespace ::= ABSTRACT-OPERATION
        INPUT     NewInput
        RESULT    NewResult
        ERRORS    { InputError,NewError,AccessError}
NewInput ::= SET {
        directory [0] Name,
        server    [1] Name OPTIONAL }
NewResult ::= UID
```

DestroyNamespace: This operation destroys the root directory for a name space.

```
DestroyNamespace ::= ABSTRACT-OPERATION
        INPUT     DestroyInput
        RESULT    DestroyResult
        ERRORS    { InputError,DestroyError,AccessError}
DestroyInput ::= SET {
        directory [0] Name,
        server    [1] Name OPTIONAL }
DestroyResult ::= NULL
```

UNS's**SetDefaultNamespace**: This operation sets the default root directory for the specified server. The default directory is used when a name resolution is started.

```
SetDegaultNamespace ::= ABSTRACT-OPERATION
        INPUT     SetDefaultInput
        RESULT    SetDefaultResult
        ERRORS    { InputError,SetDefaultError,AccessError}
SetDefaultInput ::= SET {
        directory [0] Name,
        server    [1] Name }
SetDefaultResult ::= NULL
```

The operations described above are not necessarily complete nor essential for a large distributed name service, but represent a set of proposed interaction mechanisms to be used by managers and clients when interacting with the UNS. Different languages and system support mechanisms may be used for simplicity or efficiency when implementing the interface.

3.5 Other Issues

Heterogeneity

In contrast to the Stanford approach [Mann 87] which aims to interface local name servers to a global name service, the UNS approach aims to provide a single global service. The UNS also allows different naming conventions and service architectures to be accommodated within the global system.

To enable heterogeneous local name servers to work cooperatively under the umbrella of the UNS may be seen as a functional extension to global name services such as the GNS, or the X.500, which was designed as a homogeneous or standard directory service. Users of the UNS should not be forced to adopt a single naming convention at the lower level of the UNS naming hierarchy; resolution of an IN is not necessarily a part of the global name service.

Allowing heterogeneity in a name service can be problematic, however. One problem is how to distinguish an unbound name from foreign names encountered by a name server; another is how to incorporate those directories, whose structure is dependent on a native naming architecture, into the global name service. The first problem is subtle if the DN part of a name is heterogeneous, because a name which can not be found in local contexts may be bound in a remote context. In principle, provided that any name server in a global name system knows all naming conventions, erroneous names then can easily be distinguished from foreign names. If no such server exists then there is no way to discover unbound names until all possible contexts have been exhausted. In practice, however, the space required to store naming conventions, the performance of the parsing mechanism as well as simplicity of the user interface must be considered. In a centralised naming system, it is easy to recognise an erroneous name because a server needs only consult the contexts that it maintains. As mentioned in the last paragraph, the UNS allows a single naming convention for DNs, and leaves INs to their local authorities to deal with.

The problem of accommodating directories into the global name service can be solved if local names are structured and each component names a context.[5] For example, suppose that an IN looks like "x.b.c", and the local name resolution starts from right to left.[6] Before "c" joins the global name space, the UNS name for entity "x" may be (#123/E/F/G, x.b.c). The UNS resolves the first part - DN, and leaves the second part to be resolved by a local name service. After "c" has joined the global name space, its new name is (#123/E/F/G/c, x.b). If working context is used, it is easy for the local service to resolve "x.b". "#123/E/F/G/c" may be treated as a newly added directory name in the UNS. Every thing goes well after all. People may argue that the name for entity "x" has been changed from (#123/E/F/G, x.b.c) to (#123/E/F/G/C, x.b) after c's exportation to the context "#123/E/F/G". The novel aspect of *exporting* is that only a new name - (#123/E/F/G/c, x.b) is introduced to entity "x", while its other names, i.e. full names such as (#123/E/F/G, x.b.c), or short names such as "x.b.c" still work. On the other hand, a local name server may refer to "#123/E/F/G/c" through the UNS, for it is now globally resolvable. Using different *separators* [7] does not place any difficulty to directory exporting, as explained above. For non-structured names, for example, a name to be

[5]The conditions are stronger than necessary because a multi-component name instead of a single name may be used to refer to a naming context; in this case, if it has a different separator form , the context can be easily exported. For instance, suppose the context name is "A&B"; the context can be exported to the UNS, and named as "#123/X/Y/A&B".

[6]The UNS resolves a name from left to right.

[7]A structured name is also defined as a multi-component name, which is constructed by one or more simple names (component) separated by a character called a *Separator* or a *delimiter*, such as "." or "/".

resolved by a hash function, there is no simple way to extract the name for a context to be exported.

Federated Naming

Existence of heterogeneity in a global name service is different from, although similar in some aspects to, federated naming. The APM federated naming approach [Linden 90] concentrates on "freedom of association". However, the UNS approach emphasizes the provision of a large scale, consistent, globally accessible name service for public access. The UNS allows some degree of heterogeneity, particularly on the lower levels of the naming hierarchy. The APM approach tends to give the naming authorities greater autonomy. In order to achieve global coherence, the UNS hides the differences between naming systems on the higher levels of a naming hierarchy. The UNS approach can be viewed as the ultimate form of federated naming.

Chapter 4

Dynamic Construction of Name Spaces

4.1 Introduction

If the development of the Domain Name System is reviewed, it is not difficult to find that in its early days, the ARPA name space implemented as a simple text file was maintained by a centralised name server. As the system grew, it was logical to partition the name space so as to allow local control of local naming data; this led to the current hierarchical name space maintained by a distributed name system. The DNS system makes no distinction over names, i.e. any entity name has the same structure. The major problems with this are: firstly, it lacks the ability to allow name space to be implemented according to characteristics of the entities it names; secondly, there is little flexibility to enable the name space to grow or to be re-organised as required. Similar problems exist in many other naming systems. As systems expand, it is required that name spaces be restructured so that entities in them can migrate without changing their names; it is also required that name spaces be allowed to grow upwards by merging existing ones under a new root. It is challenging to design a naming system which will continue to evolve in order to meet organisational or operational changes.

It was not until the GNS [Lampson 86], that the distinction between context names and individual names within the context was made. An example GNS name is (*ANSI*/DEC/SRC, Lampson/Mailbox); the first element denotes the SRC directory; the second element denotes Lampson's mailbox. This division mainly reflects the radically different implementation of the two parts. The GNS directory tree implementation permits its name space

to be reconstructed if required. This is intended to be done by using unique directory identifiers (DIs). However, DI resolution is not generally supported by the GNS. Since multiple name spaces may exist, the first part of the name division can be used to indicate in which name space the second part is looked up. Despite the intention of re-organising name spaces, the GNS suffers losing traces of directories when the system becomes larger. As regard to multiple name spaces, there is no reason to assume that each prospective name space to join the GNS should have exactly the same naming syntax, as demonstrated in [Lampson 86].

In this chapter, further development of dynamic name space construction by using unique identifiers, a name resolution model for resolving UDIs, server names and distinguished names are explained. The UNS name spaces defined in Chapter 3 are implemented as a logically centralised but physically distributed name space, in which two kinds of name spaces coexist, one is flat and the other is hierarchical. The design proposes a more effective name management scheme which forms the basis of the two-class name service infrastructure to be introduced in Chapter 5.

4.2 Name Resolution Model

In this section, given the definitions in Chapter 3, the UDI resolution model is explained. Technical aspects to implement a fully migration and organisation transparent global name system using unique identifiers are examined; decisions of implementing the top part of the UNS name space as a global UDI name space and allowing the bottom parts to be implemented in a local preferred manner are made.

4.2.1 More On Name Spaces

In Chapter 3, the UNS name space is defined as a group of rooted, acyclic, directed graphs, which are disjoint. In this section, a *flat* name space is defined which contains only a list of UDIs. Figure 4.1 shows the two kinds of name spaces coexisting under an imaginary super root.

The *global UDI name space* is defined as a subset of UDIs of the flat name space. The reason to set up such a name space will be explained later in this section.

Readers may have already noticed that at the beginning of this chapter, the Domain Name System was taken as an example to explain why the name space has to be partitioned and distributed among servers. It seems that introducing a flat name space to the UNS

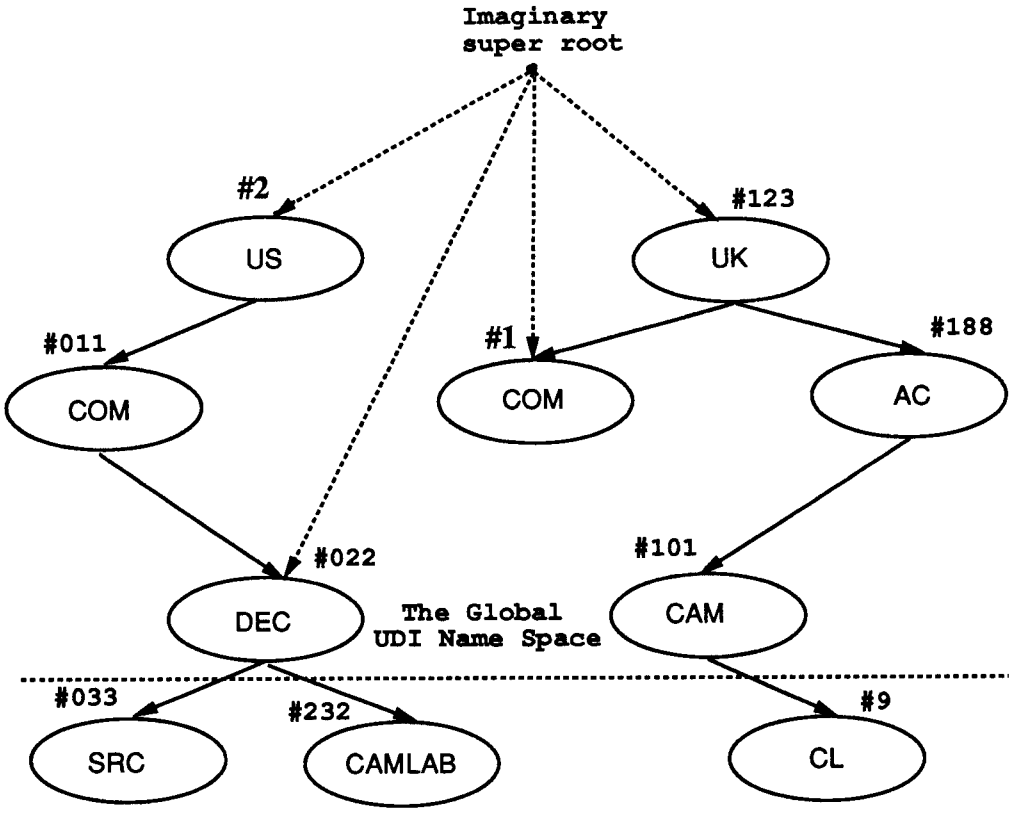


Figure 4.1: The UNS Name Space

is doing just the reverse. There is little doubt that a hierarchical name space is advantageous when handling many entities. For example, naming conflicts can be easily avoided and great autonomy is maintained in a hierarchical name space. However, it is disadvantageous in supporting flexible name space restructuring. For example, the Domain name "xx.lcs.mit.edu" will no longer be a valid name if a "us" domain is to be made the system's new root. Although flat name spaces enable name resolution to be carried out in one single step, so as to bring efficiency to the system, it is also obvious that using only flat name spaces, such as the ARPA name system, cannot work because nowadays many systems are too big to manage in that way. In the UNS, the two name spaces are combined so that both scalability and flexibility can be achieved, which is not possible if any single name space is used. More demonstrations of name space restructuring will be given in the following sections.

4.2.2 UDI and USI Resolution Model

Generally speaking, the advantages of unique identifiers are: they are location and organisation independent; they can be resolved by simple algorithms; they are relatively shorter than text names. Although unique identifiers are useful in many occasions, e.g. in authentication, in this chapter, it is focussed only on how to use unique identifiers to achieve location and organisation transparency when the UNS is designed. Location transparency means a name is not location dependent, so that moving entities does not affect their names. Organisation transparency means naming does not rely on organisational structures, so that if an entity changes affiliation, its name does not change as a result. For example, the name "#188/Cam/CL" does not indicate the network address of the entity, but hints that the entity is a directory for the Computer Lab. at Cambridge. Another example, suppose that the name "#9" is the UDI for the CL directory in the above example, as people can see that it commits itself to nothing but a number. No matter whatever possible changes are, such as moving CL's location over networks, changing its authority from Cambridge University to Oxford University, "#9" can still be used to identify the same entity.[1] Given the definitions above, there exist two kinds of name space. One consists of a group of rooted, acyclic directed graphs; the other is a flat one with a hypothetical super-root, which consists simply of a list of all UDIs. Each of the two name spaces is a complete mapping of the other's as far as the UNS directory graph is concerned. However, the top part over the dashed line in Figure 4.1 shows a possible construction of a partial UDI name space - the *global UDI name space*, in which only supreme nodes, mainly UDIs at the top most position in each naming graph, are reflected by the global UDI name space.

Setting up the global UDI name space indicates that although using a flat name space will bring great flexibility of re-organising name spaces there is also a risk that the flat name space becomes too big to manage. Is it necessary that all directories, no matter where their positions are on the graph, be given a UDI so that any directory is relocatable? Since UDIs are global unique identifiers, it is necessary for each directory to have a UDI. If only locating is concerned, the answer is no. Assigning to every directory a unique identifier which is resolvable globally will allow freedom of constructing name spaces, but it is done at the cost of performance declining; furthermore, scalability becomes a problem. It seems neither necessary nor practical to do so. In the UNS, every directory entity is named by a UDI besides its distinguished name, though only a set of those UDIs, also defined as GUDIs (Global UDIs) are put into the global UDI name space.

[1] It may be argued that some entities' names are abandoned after location or organisation changes. However, it is assumed in this thesis that old names are still used.

Assuming that the number of directories and servers to join the global UDI name space is not more than several millions, and database change rate is very slow,[2] a global index is introduced to resolve GUDIs in the global UDI name space. The name resolution is very simple and straightforward, in spite of its important role in the UNS. In the index, there are mappings between GUDIs to Unique Server Identifiers (USIs). A USI is then mapped to the server address also by the index. It is possible to map GUDIs directly to their server addresses without using USIs. It concerns tradeoff between performance of enquiries and updates. If a USI is not used, to move a server, which implies possible changes to all bindings of GUDIs to server address, will cause every entry containing the server information to be modified. If a USI is used, more lookups are needed to find a server's address. Since USIs can be resolved by a single step in the index, and hence GUDIs can be resolved in two steps, the cost of lookups is insignificant.

Global index entrances should be cached widely and rarely changed. The index is distributed globally according to internet topology and the reliability of communication links. USI may be considered as a special kind of GUDI. As to implementation, a group of servers will be responsible for maintaining the index and those servers are well known at the name service administration level.

In summary, there are three levels of name mappings in the global UDI name space. At the first level GUDIs are mapped onto distinguished names which are names relative to some roots that are children of an imaginary super-root; at the second level, GUDIs are mapped onto USIs (server identifiers); at the third level, server identifiers are mapped onto network addresses. The multiple-level naming abstraction results in properties such as location transparency, naming transparency and migration transparency. Location transparency means that the location of a directory or a server is not reflected by its name. Naming transparency means that changes of the name space, which are made in order to respond to organisational needs for instance, will not cause the old name to be abandoned. Migration transparency means that the effects of server or directory moving from one location to another are hidden.

4.2.3 More On Resolving Distinguished Names

In Section 3.3.2, the general resolution model of UNS names was discussed. However it was not mentioned how to derive a new context from the one being currently accessed. In the UNS, once a component of a name is stripped off from a distinguished name, it is

[2]Observations on the name graph show that entities sitting on the top level have a slow update rate than those on the bottom level.

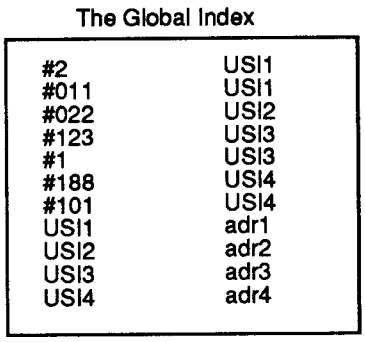


Figure 4.2: The Global Index

looked up in the current directory. If there is a match, then a server name is returned. The server name must be resolvable either in the same context, or in a context which is *closer* to a global context. For instance, assume that the current context is #101 (Cambridge directory). If given the name "#101/CL/SRG" to resolve, looking up CL in the current context will get a server name "#101/serverA"; to look up "serverA" in the same context will get the address attribute of "serverA"; now the current context becomes "CL". This procedure is repeated until the address of a server keeping "SRG" directory is found. Note that in order to avoid name resolution loops, the distinguished name for a name server should be shorter than that of any entry in any directory the server stores - this is called a shorter-server-name approach. On the other hand, if a server does not store a directory whose name starts with a GUDI, the server must know another server that either has a directory with a GUDI or is "closer" to such a directory. This is to ensure that the global UDI name space is reachable and no loops will be formed during name look-up. In the next section, some invariants to guarantee that:

1. server navigation is well constructed so that any context is reachable.
2. to resolve a name will not, at some stage of name resolution, get a handle which is the same as the name itself - producing a name resolution loop.

will be described in a more formal way.

In the discussion above, it was assumed that there was a current context available. How to get such a starting context remains unknown. In fact any name resolution starts from a global context, which is one of the directories whose UDIs are in the global UDI name space. Entrance to the global index is well known, and the index itself is widely replicated or cached in the UNS. Caching is used to improve performance. For instance, the result of looking up "#101/CL/SRC" may be cached to resolve any name within the same context.

Servers will cache their own names so that they do not have to send them to the global index for resolving.

4.2.4 Naming and Locating Name Servers

How to name and locate name servers is a very interesting issue. Should servers be named in the same way as other entities, or should they be treated specially? Considering that name servers may be on machines that are not only for naming but also for running other services, i.e. several different services are coexisting on the same machine, it is necessary to name all servers. To give a name server a name also enables location of the server to be changed transparently.

There have been three ways to tackle server naming. The first method is to set up a global context for keeping all server information. For instance, a registration database was used by Grapevine to name registration servers, message servers and registries [Birrell 82]. The database was widely replicated, i.e. every registration server of Grapevine knew the names and addresses of all message and registration servers, as well as names of all registries and all names of replication servers of those registries. Managing a system with a maximum sites of 30 Grapevine servers and a total load generated by 10,000 users did not seem to create scaling problems. However, going beyond the specification of Grapevine created problems. One problem was that the resource location algorithm selected the nearest up server from among those providing the resource in question. This would become too expensive if a large number of servers were involved. It would also become more difficult to have such a global context stored by every registration server. Another potential problem could be that the cost of propagating updates became too high due to the heavy replication of the service configuration data, as it was shown later by Clearinghouse [Demers 87] which is the successor of Grapevine; although not like Grapevine, the configuration database has been partitioned by introducing one more level of naming hierarchy.

The second method is to use the name service itself to store and to resolve server names. For example, in the GNS [Lampson 86], resolving a directory name will get a server name for that directory, so the next task is to resolve the server name. If a directory stays on a server whose name contains the directory name, a loop is formed. Thus restrictions must be introduced to the GNS in order to create proper server names. To use the name service itself to resolve server names is carried out at the cost of putting more constraints on naming and replication. There are three constraints (see [Lampson 86] for detail):

S1 Every directory d on a direct path from the root to an entry that stores a server

address has $d.inSN = true$.

S2 If $d.inSN = true$, then either d is the root , or a copy of d is stored on a server with a name shorter than any direct name of d.

S3 Every server s either stores the root, or $s.up$ is a shorter name of another server, and s stores a copy of the directory for $s.up$.

These invariants guarantee that the root of the directory tree is reachable by starting at any name server. It prevents the loop that looking up a name server would in turn require looking up the subject directory on that server. However by naming servers and resolving server names in a similar way as that of other entities, the GNS avoids linear growth of the configuration data as system size increases. It also improves the performance of updating configuration data.

The third method is to treat configuration information as *knowledge*, which is attached to a context to indicate a possible server holding the next context on a name resolving chain. An example of such systems is X.500, in which locating name servers is done by using a special data structure called *knowledge*. A X.500 system is composed of a number of DSAs (similar to name servers), each of which is responsible for a group of fragments which are subtrees of the global directory information tree. Navigation requires that each DSA be aware of the responsibilities of other DSAs. The awareness is called knowledge. A reference is defined as entries which associates the name and the address of a DSA with names of fragments for which the DSA is responsible. Knowledge is implemented as a set of references. For example, the reference describing the responsibility of a DSA could be: [3]

```
{dsa:name of DSAn,
 address of DSAn,
 responsibilities:"/c=a/o=c/u=f","/c=a/o=c/u=g"}
```

Minimal knowledge of a DSA can then be defined as:

```
MinimalKnowledge::=SET{
self_dsa Reference,
immediate_superiors SET OF Reference,
immediate_inferiors SET OF Reference}
```

[3] For in the standard itself, knowledge reference is not very clearly represented, the example is taken form [Benford 88], in which, ASN.1 is used to represent knowledge reference.

Minimal knowledge is essential for navigation to occur; as can be seen in the example, minimal knowledge for a name server contains its immediate superior server's reference as well as its immediate inferior server's. In [Benford 88], a method for maintaining such knowledge is proposed. However, it is argued in [Kille 89], which describes the UK pilot X.500 project, that there is no need to have additional information such as *knowledge* to deal with a name service's own configuration.

So far we have seen three methods of naming and locating name servers. The first method is simple and easy to be configured or re-configured, but it does not scale properly. The advantage with the second method is no special method for maintaining configuration data is necessary. However, the disadvantage is that it enforces a set of rigid rules for server naming and directory replication. The third one is that server information is treated as special naming data. Particular methods are needed to deal with the special data, and it could lead to a complicated design.

In the UNS, name servers are named by the service itself, but they are treated slightly differently from individuals. Individuals cannot appear in any directory in the UNS name space but a name server entity can. A **Reference** is defined as a set of server name to address mappings which are kept by a name server for proceeding with navigation. In the absence of replication, a reference is essential for the navigation to proceed. With the help of caching or replication, the usage of references may be reduced a great deal. This will be further discussed in Chapter 6.

In order to implement the UNS name resolution model, there are three server invariants that should not be violated when running the service.

Definition4.1 A context $\mathcal{A}$ on a naming network **N** is said to be *closer* to a global one $\mathcal{G}$ (represented as $\succ_{\mathcal{G}}$) than another context $\mathcal{B}$, if and only if its direct path name from $\mathcal{G}$ is shorter than that of $\mathcal{B}$. This can be represented as: $\mathcal{A} \prec_{\mathcal{G}} \mathcal{B}$ iff $P_{\mathcal{G}}(\mathcal{A}) < P_{\mathcal{G}}(\mathcal{B})$, where $P_i(j)$ defines the direct path name from i to j. Similarly, suppose the name server s_1 storing $\mathcal{A}$, s_2 storing $\mathcal{B}$ and $P_{\mathcal{G}}(\mathcal{A}) < P_{\mathcal{G}}(\mathcal{B})$, therefore $s_1 \prec_{\mathcal{G}} s_2$.

Definition4.2 A server s is a *root server* iff $\{\exists \mathcal{G} \in \mathbf{G} \mid \forall s_j \in \mathbf{S},\ s \preceq_{\mathcal{G}} s_j\}$, where **G** denotes the set of global contexts, and **S** denotes the set of all name servers comprising the UNS, $\prec_{\mathcal{G}}$ means *closer* to $\mathcal{G}$ and $\preceq_{\mathcal{G}}$ means *closer* or *equally close to* to $\mathcal{G}$. [4]

The UNS server invariants :

[4]Note that for any directory with a GUDI, it is defined that $\{\forall \mathcal{G}_i \in \mathbf{G} \mid \forall \mathcal{G}_j \in \mathbf{G},\ \mathcal{G}_i = \mathcal{G}_j\}$. This definition indicates that any two global contexts are equally *close* to each other.

r1 If $s \in \mathbf{S}$ is not a *root server*, its distinguished name should be shorter than the distinguished names of any entry in any directory it stores.

r2 If $s \in \mathbf{S}$ is not a *root server*, it must store references to other name servers which are either *root servers* or are *closer* to $\mathcal{G}$.

r3 If $s \in \mathbf{S}$ is a *root server*, it must contain the global index.

r1 guarantees that no name resolution loop is formed by naming servers. **r2** ensures that a given name is either resolvable or found unbound in case it is invalid. Usually in a hierarchical name space, if the root is reachable, then all names in the name space can be resolved. However, the UNS name space is composed of several disjoint name spaces. The global index has been introduced to allow names in all name spaces to be resolved starting from any name space. Within a name space, its root (one of the global contexts) can be reached, which is assured by **r2**. **r3** ensures that if a global context can be reached, all global contexts can also be reached. A name server will cache name to address mapping of itself so that it will not have to look up its own name. Maintenance of service configuration data will be discussed in Chapter 6.

An alternative approach to guarantee that a global context can be reached starting at any server is to store the global index on every UNS server - a heavy replication approach. Although it improves the efficiency of name resolution, it makes update propagation more difficult even though the change rate of the index is small. However, a weak-point of the former is that, besides more costly name resolution, the complexity of configuring the name service is increased if compared with the heavy replication approach. However, the performance of propagating updates can be improved because a particular set of the configuration data is only replicated on a much smaller scale. For instance, only those servers which store the child directories will need to keep references to some servers which have the parent directory.

Suppose that the maximum depth of the naming network is not greater than 5, e.g. each level on the network (a tree) represents correspondingly the imaginary super root, countries, organisations, units within an organisation, and groups within a unit. There are about $200 \times 200 + 200$ entities at three top most levels, $30 \times 20,000$ at the forth, and $10 \times 600,000$ at the fifth. It is also assumed that there are about 100 root servers, 100×200 organisational servers and $100 \times 200 \times 30$ unit servers(e.g. every two countries share a root server, every organisation has an organisational level server, and every unit within an organisation has a unit level server). If the global index is replicated everywhere, then 620,100 copies exist. Each time the index is updated, at least 620,000 messages are created on the internet. If the index is only replicated among organisational level servers, then

20,100 copies exist and less than 30 time messages are produced per update.[5] Regarding name resolution cost, only one more time indirection will be introduced if compared with the heavy replication approach. This analysis suggests that communication network traffic may be significantly reduced by using references instead of heavy replication. How much complexity this introduces will be discussed in Chapter 6.

Server name resolution can be carried out as was discussed in Section 3.3.2 and Section 4.2.3. Navigation is implemented by making the server name resolvable no matter where name resolution begins.

4.3 Examples of Dynamic Name Spaces Construction

Given the definition of the abstract operations in Chapter 3, the ability of the UNS to merge or restructure name spaces is demonstrated in this section.

4.3.1 Growth

Merging two existing name spaces is illustrated in Figure 4.3. Suppose there are two name spaces **B** and **C** with the root directory B and C respectively. The GUDI #121 is assigned to B when B is created; #131 to C. Before the merging takes place, DNs look like $\#121/n_1/.../n_m$ in the name space **B**; similarly, $\#131/n'_1/.../n'_l$ in **C**. After the merging, a new root directory A is created with GUDI #111; A, B and C are all members of the GDUI Name Space, which is flat. Any DNs such as $\#121/n_1/.../n_m$, $\#131/n'_1/.../n'_l$, $\#111/B//n_1/.../n_m$, and $\#111/C/n'_1/.../n'_l$ are resolvable names in name spaces **A, B** and **C**. Operations involved are: **NewNameSpace, MoveDirectory, InsertUDI.**

4.3.2 Restructuring

Moving a directory is demonstrated in Figure 4.4. Suppose directory B and C are child directories of directory A. Now C is to be moved as a child of B. Before the move, any DN containing C may look like $\#111/C/n1/.../n_l$, but after the move, both $\#111/B/C/n1/.../n_l$, and $\#111/C/n1/.../n_l$ are valid names. The DN $\#111/C/n1/.../n_l$ is replaced by the soft-link $\#111/B/C/n1/.../n_l$ or $\#131/n1/.../n_l$ if C's GUDI is kept in the GUDI name space. Operations involved are: **ModifyEntry, MoveDirectory.**

[5] Updating references may cause approximately 200 messages to be sent, which may be ignored.

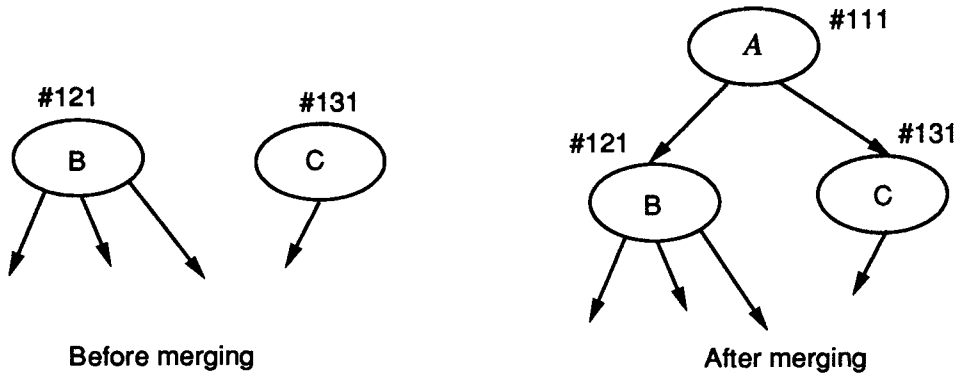


Figure 4.3: Restructuring the name space by merging name spaces

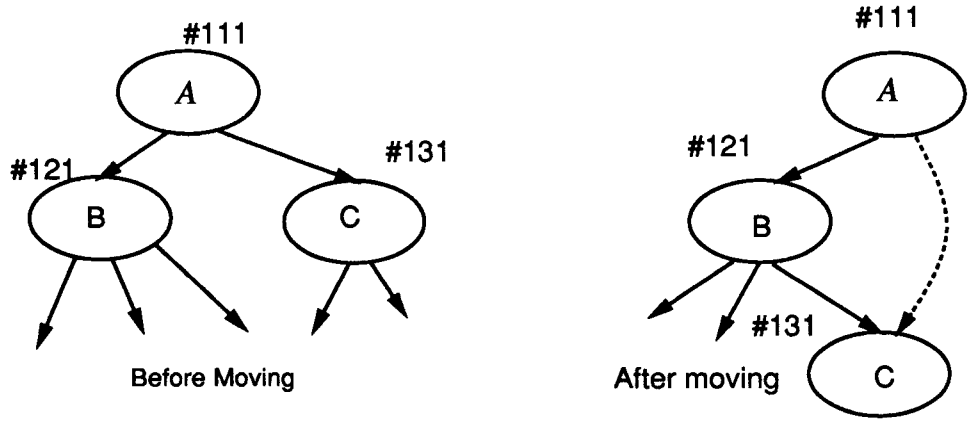


Figure 4.4: Restructuring the name space by moving a directory

Chapter 5

Maintenance of the Naming Database

5.1 Introduction

In this and the following chapter, name service maintenance, in particular the replication control and dynamic service configuration, is discussed. Maintenance of the UNS largely relies on the protocols of consistency control. Although it is widely believed that loose replication control mechanisms are good enough to satisfy name service requirements, after comparing several replication control mechanisms, and exploiting naming system semantics, it is found that better maintenance can be achieved by using not only a loose replication control but also a tight one. In fact, it is very difficult for any single replication control method to meet all the requirements for maintaining naming data. Therefore, a two-class name service infrastructure is proposed.

The infrastructure emphasizes that different methods of replication control should be used to reflect different semantics of naming. Loose replication control algorithms simply will not suit all naming requirements. For instance, considering a particular set of data in a large distributed naming system, it is valuable to allow those clients who are more interested to be able to get the most recent changes. On the other hand, if naming data is heavily replicated, it is expensive and unnecessary to force all replicas to be consistent immediately; it is less satisfactory to make them consistent eventually only - provided no more updates will occur, consistency will be finally reached. So, new methods are required to ensure eventual consistency as well as to find out, in a timely way, what is current.

In the following sections, the newly proposed consistency control mechanisms are intro-

duced. Section 5.2 reviews a number of mechanisms that can be used for maintaining naming database. Section 5.4 describes the two-class service infrastructure. Section 5.5 gives details of the protocol developed for the first class service. Section 5.6 explains how updates are propagated among servers, and finally Section 5.7 discusses how caching is used to improve performance.

5.2 A General Review of Replication Algorithms

A brief review of some well-known replication control algorithms is given in this section. The intention is to outline the advantages and disadvantages with respect to naming database maintenance, to investigate the possibility of them being used for naming systems, and to explain the necessity of developing a two-class name service infrastructure.

5.2.1 Primary Site

Primary-site methods have been developed for replication control of distributed databases [Alsberg 76, Stonebrake79]. The original method worked roughly as follows. One replica is designated the primary and the rest are slaves. Different objects may have different primary sites. Each update should be applied to the primary. The primary site is responsible for informing all, or a majority of, slaves. On receiving the new updates, a slave sends an acknowledgement to the primary; the primary waits for sufficient number of acknowledgements to decide whether to commit; if it decides to, it informs the client; otherwise it decides to abort, each site should back out the transaction.

QUIPU [Kille 89] employed an algorithm of this sort, but in an asynchronous way. For instance, QUIPU defines a master DSA which maintains a master copy of a certain portion of naming data, and also defines a number of slave copies stored by other authorised DSAs. When an update is done on the master copy, the master DSA will propagate it to all slave copies if requested.

The primary-site methods suffer from a centralised control, and possible bottleneck at the primary site. It is also vulnerable to site or communication failures.

5.2.2 Quorum Consensus

Quorum consensus algorithms [Bernstein 87] such as weighted voting [Gifford 79] can be used for general replication control. The idea is to assign to each replica a number of

votes for reads(r) and writes(w), and the sum of r and w should be greater than the total number of votes(v), i.e. $w + r > v$ and $w > v/2$. A transaction must collect a read-quorum before any read, and a write-quorum before any write. This method is more robust than primary-site, because there is no centralised control. Applications can adjust availability of reads and writes by varying the number of votes assigned to each replica. Quorum consensus can work well with network partitions or site failures, but it pays for its resiliency to communication failures by increasing the cost of read and by increasing the required degree of replication. This cost is high if communication failures are infrequent. It would be preferable if the cost could be reduced during reliable periods of operation, possibly at the expense of higher overhead during unreliable periods. Few naming systems use voting, but file systems such as [Bloch 82, Gifford 79] do, and an underlying multi-site atomic action environment is assumed in both cases.

The Paxon part-time parliament protocol [Lamport 89] does not need the support of multi-site atomic action although it uses voting too. It has fewer restrictions than algorithms such as weighted-voting with respect to replication controls but is less generic because it does not support the transaction abstraction. It requires up to $5n$ message exchanges and 5 rounds. An optimised Paxon protocol has only 3 message delays and about $3n$ messages. The Paxon protocol does not tolerate arbitrary, malicious failures, nor does it guarantee bounded-time response. However, consistency is maintained despite the failures of any number of processes and communication paths. Thus, the Paxon protocol is suitable for systems with modest reliability requirements that do not justify the expense of an extremely fault-tolerant, real-time implementation. Later in this chapter, a UFP protocol based on the Paxon protocol is developed for the UNS.

5.2.3 Commit Protocols

The two-phase (2PC) and the three-phase (3PC) commitment protocol are well known in distributed database management systems [Stefano 87, Bernstein 87]. The protocols are performed by a distributed transaction manager (DTM), called *coordinator*, together with a number of DTMs, called *participants*. The basic idea of commit protocols is to make a unique decision for all participants with respect to committing or aborting all the local operations. 3PC is resilient to site failures only. With network partitions, however, it is possible to process transactions within a partition which includes the primary copy or a majority of sites of all data needed. 3PC is non-blocking except for a total site failure; in absence of failures, up to $5n$ messages are exchanged; the time complexity is 5 rounds of messages. 2PC can tolerate communication failures as well as site failures. 2PC is cheaper

if time or message complexity is concerned, but it is subject to blocking.

So far there have been few attempts to use the two-phase or the three-phase commitment protocols for name services.

5.2.4 Sweep

Sweep is used for update propagation of the GNS [Lampson 86]. To be reliable, all replicas are linked into a virtual ring. Updates are allowed to any available replica. A sweep updates all replicas. A query result is guaranteed to be good up to the time of the last completed sweep.

If collecting updates is done in parallel, the time complexity of a successful sweep is 3 rounds of messages; $3n$ messages are exchanged. The method is not resilient in the presence of any failures. In addition, maintenance of the virtual ring is subtle. If the ring is broken due to some failures, a new ring must be reformed by using directory references or information provided by the administrator. Since the ring determines the current content of the database, reforming a ring is never done automatically, but must be controlled by an administrator. There is no guarantee that the current state of the database can be found. A snapshot always gets back the database state consistent up to the last-sweep-time together with an arbitrary set of new updates.

5.2.5 The Epidemic Algorithms

In [Demers 87], several randomised algorithms such as Anti-entropy and Rumour mongering for replicated database maintenance are analysed. These algorithms are very simple and require few guarantees from the underlying communication systems. Provided that there are no more new updates, the algorithms will bring the database into consistency eventually, even when there are failures. They generate reasonable traffic for the underlying network. These methods have been run with the Xerox Clearinghouse name service successfully.

5.2.6 Lazy Replication

Many services provided by computers must be highly available. By taking the semantics of a service into account, implementation constraints on replication can be weakened. Lazy replication (see client-ordering in[Ladin 90]) allows an update operation to be performed

in a single replica, and for the effect of the operation to be passed to other replicas asynchronously. The propagation of the effect is carried out by exchanging *gossip* messages, which contain the latest updates. The frequency of gossip may vary from one application to another. A query comes with a multi-part timestamp which is a combination of timestamps for each replica. A multi-part timestamp can be used to check if a replica has the information concerned by a query. Since extra information (i.e. the multi-part timestamp) is needed for ordering, the algorithm is good provided that the size of such additional information is small (i.e. there is a modest number of write-replicas) and provided most operations can take advantage of the laziness (i.e. client-defined ordering is appropriate). The method appears to be applicable to a wide range of applications, including garbage collection of objects in a distributed heap [Ladin 88], locating movable objects in a distributed system [Hwang 88], and so on.

5.2.7 Summary

The replication control mechanisms reviewed above can be divided into two categories: those giving immediate consistency, such as the primary site, quorum consensus, commit protocols and Paxon, are synchronous algorithms; those for long-term consistency, such as the epidemic algorithms, sweep and lazy replication, are asynchronous ones.

Synchronous methods place restrictions on either query or update operations, or on both, for maintaining one copy serializability. Although their implementations differ and semantics can be used to release some restrictions under particular circumstances, they are tightly-coupled and do not scale very well. It is also possible to lose locality of reads. As discussed in the last section, for a large distributed and replicated naming system, it is very expensive to run synchronous consistency control protocols; various autonomy requirements can make such protocols impossible to implement.

On the other hand, provided only with the guarantee that a piece of naming data is good up to some past time, a client will not get the necessary help if the data fails to work. The problem with sweep, lazy replication or the epidemic like algorithms is that there is no way to find out what is wrong until either another sweep is done [1] or an unknown time. [2]

[1] Although a client initiated sweep may be allowed, it is expensive to carry it out often. The Sweep algorithm does not scale well, and updates may be lost too, as mentioned in Section 5.2.

[2] The epidemic methods imply that the database will become consistent at a future time, which can not be precisely specified.

5.3 Semantics of the Name Service

In this section, assumptions about naming system semantics are made. There is always trade-off between the accuracy of the naming data and its availability. It is common to see that the naming data is maintained in a loose way; there are a number of good reasons for doing so:

- The rate of naming data update is slow
- The data can be treated as hints, which are explicitly permitted to be wrong
- Most updates are made by system administrators

The first assumption is basically true, but in many cases, the rate of change to the naming data is not as low as one would expect. For instance, British Telecom makes approximately 5000 updates a week out of 1 million entries in the region containing Cambridge, which suggests about a 26% annual update rate (according to a BT phone book manager). The number for some other naming systems is even higher. It is worth mentioning that not all naming data changes at the same rate. For example, a person's name may last for decades, while a process identifier lasts less than a second. Another example is that organisations will not usually change their names faster than people change their affiliations. People's interests in different naming data are also diverse.

The second assumption is also true, providing uses of obsolete naming data can be detected within a reasonable time, and the cost of recovery from using wrong data is acceptable. However it is not easy to make such claims for all possible naming applications. For example, a very important mail message could be lost due to lazy update propagation, although in systems such as the mailing system, loose consistency control is commonly seen. As noticed by Needham and Terry [Needham 88, Terry 87], both detection and recovery require efforts from outside the naming system. Figure 5.1 shows the recoverability of naming systems. Therefore, it is worth making a name service offer the most current information rather than only hints.

The third assumption is nearly true, but sometimes clients are allowed to change their data registered in a name server, e.g. passwords. It is interesting to notice that the naming data which may be changed by users is not usually replicated a great deal, while the naming data which can only be changed by administrators may be heavily replicated. For example, the global contexts belong to the latter. In some name systems such as the GNS, there is a similar feature: the demand for replication of those directories on the top levels of the directory hierarchy is much higher than that of those on the bottom levels. No

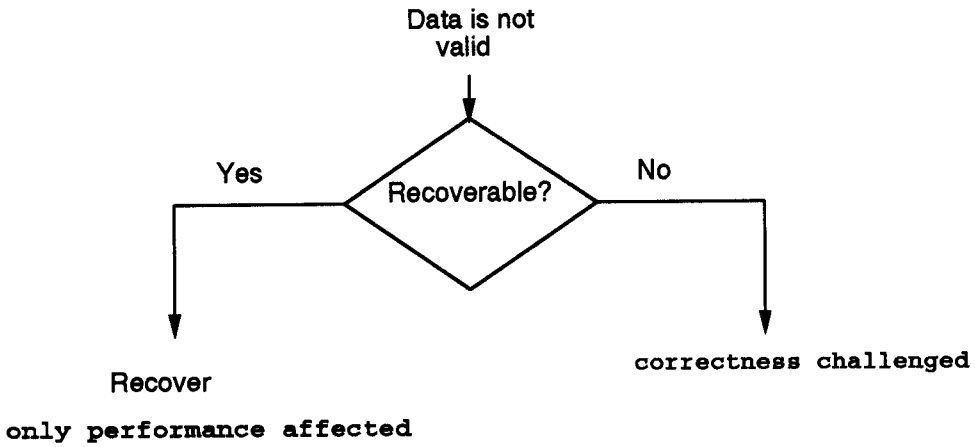


Figure 5.1: Diagram of Recoverability

matter who initiates updates, name services should be able to maintain strong integrity of the naming data.

Semantic Properties

In order to release some of the constraints on data consistency, name system semantics should be exploited. A global name service should exhibit the following properties:

- *Few conflicts*: considering a single naming data entry, there are few conflicts between queries and updates, as well as among different updates.
- *Enormous number of queries*: the number of queries made to a name service is much greater than that of updates.
- *Only addition needs validation*: validation is necessary only when a new name is to be added to the naming database to avoid conflicting names.
- *Low change rate*: although the number of updates made to a naming database is considerable, it is still less than that in file systems.
- *Few constraints*: it is good enough if a replicated name service appears to the client to provide the same observable behaviour as a single copy one.
- *Caching hints are acceptable*: when detection of invalid data on use is allowed, cached naming data can be used as hints.
- *Restricted availability*: most naming data is protected from being made public, less replication needs to be done for such data. Some naming data serves public interests, and has to be widely replicated for high availability and reliability; changes to such data is interdependent.

- *Single-shot transaction*: Naming operations are usually constructed as single-shot transactions. [3]

Requirements

Up to now, the semantics of a large name system and various issues on its maintenance have been discussed. To achieve the design goals of the UNS outlined in Chapter 3, the mechanism for the UNS maintenance should be:

- highly available
- reliable
- scalable
- fault-tolerant

5.4 The Two-class Name Service Infrastructure

In previous sections, different replication algorithms were analysed and the semantics of naming systems was given. Heavy replication is one way to provide a highly available name service; it also enhances the fault-tolerance of the service. Caching always proves a useful method to improve performance; for applications classified by a slow update rate and frequent queries, such as naming, caching is often used. Much previous work has been done in these areas, such as those found in [Lampson 86, Terry 87, Birrell 82]. However little effort has been attempted to allow clients to be able to find the most recent data from a naming database, when a cache failure (due to stale naming data) is encountered. Strong integrity of the naming data is not combined with cached information to provide a more desirable name service. An interesting observation of naming applications is that a client may like to contact a name service after a cache failure. Providing strong integrity for a naming database leads the client to find out what went wrong if a piece of data failed to work because it was obsolete. A large naming system can be implemented without trading consistency for high performance, which brings the motive to combine the tight replication control protocols and the loose replication control protocols into a new mechanism - the two-class name service infrastructure. In this section, the new method is given in detail; in

[3] A single-shot transaction is defined in [Birrell 85] as a transaction which is not composed of multiple client actions.

the next two sections, methods for the first class and the second class service are explained separately.

5.4.1 The Computation Model

A commonly used paradigm in constructing distributed systems is the client/server model. In this model, a server maintains data objects and defines operations which are exported to clients. Clients invoke these operations to manipulate the data managed by servers. Communication between servers and clients is carried out by remote procedure calls. A server may itself be a client of another server.

The computation model makes the following assumptions about system hardware and effects of failures. The system is globally distributed and consists of a collection of interconnected computer communication networks, each of which contains a group of connected physical machines. Machines and networks are considered as entities of the system. The word "machine" denotes a server machine or a client machine, which runs a service or an application accordingly. A group of server machines may provide with clients a service cooperatively, and in such case, each server machine is called an instance of that service. Entities of the system may be dispersed all over the world and may belong to different organisations or individuals who may not trust each other. A machine can fail, but it does not exhibit Byzantine behaviour. [4]

The communication network assumed in the thesis may have any kind of topology. Again, no Byzantine behaviours are under consideration. For instance, the network will not deliver corrupted messages. Network failures may be losses, or duplication of messages, network partitions, and delayed or out of order message deliveries.

Every machine has a loosely synchronised clock which is used to measure time intervals or trigger actions.

5.4.2 The Two-class Name Service Infrastructure

The basic idea of the two-class name service infrastructure, as demonstrated in Figure 5.2, is that the replicas of a directory are divided into two categories: those on servers running a tight replication control protocol - the first class servers, are called *replicas*, which are used for strong integrity of the naming database, and those on servers running a loose

[4] A Byzantine fault is said to occur when a system component delivers a wrong result. The contrast is with a fault which just causes the affected component to stop. See [Lamport 82, Schlichtin83] for details.

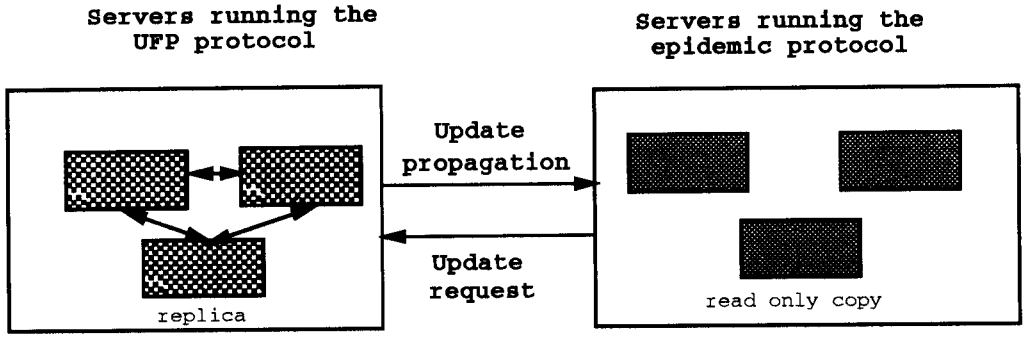


Figure 5.2: The two-class name service

replication control algorithms - the second class servers, are called *read only copies*, which are used for high availability. The infrastructure normally consists of a small number of first class servers and a large number of secondary servers. The number of each type of server may vary under different circumstances. In a situation where data is replicated locally with a reliable underlying system and communication support, it may not need any secondary servers because running the first class servers is enough. The client actions which affect the name service may be unpredictable or independent. For instance, a client changes his password in private, or a server is moved without notifying the rest of the world. In such a situation, it is desirable not to prevent the client actions from happening, thus loose replication control or less replication is preferred. There may be an arbitrary number of caches kept by clients, which may be either applications or other servers.

In order to make the service reliable only the first class servers are allowed to carry out updates. The second class servers are informed by the first class servers after a successful update. Among the first class servers, a tight replication control protocol, i.e. the UNS First Protocol (UFP), is run. Anti-entropy and call-back are run for propagating updates between the firsts and the seconds.

Besides the two-class service infrastructure, caching is also used. The major difference between read only copies and caches is that the former are treated as "official copies", which are subject to authorised refreshes from time to time; the latter are just normal caches. Applications may choose from any of the three kinds of data whichever suits them best. There is a two-way communication between the firsts and the seconds - the firsts use call-back to push updates to the seconds; the seconds use anti-entropy to poll from the firsts when necessary. Caching, as in many previous name systems or database systems, is used to improve performance. A more detailed discussion about caching is given in Section 5.7 .

The novel aspects of the infrastructure are: strong integrity of naming data is provided; availability of the most demanding service operations is high. The client of name service can find the current data in case of need. The most important fact to note about the two-class service infrastructure is that it combines tight replication control mechanisms and loose ones into a new form of service management which cannot be judged on the terms normally apply to either separate component. The emphasis here is to provide correctness without compromising high availability.

5.5 The UNS First Protocol (UFP)

The replication control protocol to be discussed in this section is for the UNS first class service. As indicated in the last section, the first class service is vital for maintaining the strong integrity of the naming data. Two major requirements to the UFP are reliability and fault tolerance. Thus, the probability of information loss and service disruption due to communication or site failures should be minimised. In order to satisfy the two requirements, the UFP protocol is developed to coordinate the first class servers for maintaining one copy serialisability (1SR) [Bernstein 87] in an efficient and cost-effective way. Quorum consensus is integrated in the UFP so that a few sites or communication failures will not stop the service.

The UFP is influenced by the Paxon protocols [Lamport 89] which are described as if they were developed originally by an ancient civilisation on the island of Paxos for their parliament passing decrees of the land. They ensured that the parliament functioned properly even in case that some of the legislators walked in or out of the chamber during the proceedings. No matter what happened, consistency of the ledgers, which recorded the decrees passed, was maintained. In terms of distributed computing, the legislators can be seen as the processors, the ledgers as the replicated data and the legislator's walking in or out of the chamber as the processor's coming up or going down.

In the rest of this section, the Paxon Synod protocol is outlined, which is followed by the introduction of the UFP. The UFP is then described in detail with some examples. Finally, possible elaborations to the UFP are discussed.

5.5.1 The Paxon Protocols

The Paxon Synod protocol was used by the Paxons for passing a single decree. Each Paxon legislator maintained a ledger in which he recorded the numbered sequence of decrees that

were passed. Requirements on the protocol included consistency of the ledgers, and the decrees were eventually passed and recorded in the ledgers.

The procedure of passing a decree was called a ballot. One of the legislators had a special role to play during a ballot, i.e. as the president. The president started a ballot by assigning a unique number to it, and then asked the other legislators inside the chamber to see whether he could start the new ballot. The legislator receiving the request might answer "yes" or simply ignore the request. Once the president got enough positive replies to his request, he proposed the decree to be passed and sent it to every legislator in the chamber. On receiving the decree, a legislator might vote for it and inform the president or again ignore it. If a majority of the legislators voted for the decree, the president would tell everyone in the chamber to write it down in their ledgers. Once informed, the legislator would write the decree in his ledger. Having missed a passed decree, a legislator would find his ledger out of date once he learned the number of the current decree being voted. He could then bring his ledger up to date by copying from other ledgers which had a larger decree number.

There are three steps in a ballot: first, to choose a decree; second, to vote for the decree if chosen; third, to record the decree in the ledgers if it was passed. In the first step, the president assigned a number which was bigger than what he ever tried before. The number was made unique: this could be done by remembering what he had previously initiated with a note in his ledger. To avoid a different president initiating the same ballot number, an obvious way is to associate the name of the president with an integer and use lexicographical ordering. The voting rule is simple. On receiving the president's first request, a legislator checked his own note to see whether the ballot number was big enough, if so, he replied with the decree he had voted for last time. Otherwise he did not do anything. Similarly, a legislator might vote for the decree chosen by the president before the second step or do nothing.

Any legislator might walk in or out of the chamber during a ballot. If a legislator left the chamber, the president would get neither a request passed to him nor a reply from him. If the president left the chamber before a ballot finished, a new president should be elected for initiating the ballots. In the beginning of a ballot, one of the legislators in the chamber was elected as the president.

In [Lamport 89], Lamport first described and proved the correctness of the Paxon Synod protocol. He then derived a multi-decree protocol, which can perform multiple updates. The Paxon protocols may be applied to distributed computing systems. For example, the multi-decree protocol can be used for making multiple updates to the replicated data. A

more formal description of the Paxon Synod protocol is given in Appendix A; see also [Lamport 89] for further details of other Paxon protocols.

In summary, the Paxon protocols work under the assumption that the legislators (processors) are fail-stop. The protocols therefore are cheaper and simpler than the protocols which are designed to tolerate arbitrary, malicious failures. Consistency is guaranteed by using a quorum. If a president gets elected and every legislator in the chamber works promptly, progress is ensured. Most importantly, the protocol does not depend upon any underlying distributed transaction support. These properties make the Paxon protocol interesting to the UNS.

5.5.2 Introduction to the UFP

Although Lamport indicated that the Paxon multi-decree protocol could be used by a name service, no detail was given in [Lamport 89]. The UFP protocol to be introduced in this section is inspired by the Paxon multi-decree protocol, which retains the advantages of the Paxon protocol while making further use of the naming semantics to improve efficiency. There is no election algorithm needed by the UFP for simplicity. Once integrated with the two-class name service infrastructure, the UFP protocol ensures the integrity of the naming data.

The UFP has the same computation model as described in Section 5.4. The unit of replication is commonly a directory or an index, although fine-granularity replication can also be used with the UFP if required. For the data which is likely to be widely replicated, the two-class name service infrastructure provides a framework to place the replicas of the data on the first class or the second class service. The novel aspect of the UFP is that it constructs a reliable and fault-tolerant first class service at a reasonable cost. No underlying multi-site transaction support is needed, so that the UFP is less expensive than other quorum-based approaches such as weighted voting [Gifford 79]. Elaborations can be made to reduce the cost further.

There are three phases in the UFP: *validation*, *preparation* and *commitment*. The protocol is for synchronisation of the first class servers. Synchronisation is an action which makes replicas of a multi-copy directory or index reflect the client updates in the same order. One of the servers involved in a synchronisation serves as the *coordinator*, which is responsible to start and organise the synchronisation; the other servers are *participants*. A client request can be sent to any of the servers. Thus any server can start a synchronisation. On receiving a request, the server becomes the coordinator of a new synchronisation and

enters the validation phase. The coordinator commences by sending a message to every server keeping the replica to check whether it has the current data. If a quorum of the participants agree on the data and vote for it, the request can then be chosen. The protocol goes on to the preparation phase where chosen request is sent to every server in the quorum. If again a quorum is formed, the client request will be implemented in the commitment phase.

Each participant runs in either one of the two modes: *normal mode* or *recovery mode*. A server usually stays in the normal mode unless inconsistency has been discovered. If a synchronisation succeeds, an *ack* message is sent to the client by the service.

Before the UFP is given in detail, the following definitions are made:

The Replica Set

In the UNS, the unit of replication is either a directory or a global index.[5] A directory may be replicated on several servers; each server stores a *replica*. Let d denote a replicated directory, r denote a replica, and $\mathbf{S}$ denote the set of all the servers comprising the UNS service. Let $\mathbf{R}_d$ denote a replica set, which is defied as $\mathbf{R}_d = \{r \mid r \text{ is a replica of } d\}$. Let $\mathbf{S}_d$ denote the name server group (NSG) in which each server is a physical storage site for a replica of d, i.e. $\mathbf{S}_d = \{s \mid \exists\ r\ \in \mathbf{R}_d\ \wedge\ r\mathbf{Ons} \text{ is true}\}$, where $r\mathbf{Ons}$ is true means r is stored on the server s. Individual replicas are not normally visible to the client. All servers in the same NSG offer the service cooperatively.

For example, in Figure 5.3, there are three replicas A, B and C for d. Thus $\mathbf{R}_d$ = { A, B, C}, and $\mathbf{S}_d$ = { S_A, S_B, S_C }, where the server S_A stores replica A, S_B stores B and so forth. The naming database is partitioned and distributed among the UNS servers so that there are many NSGs for different data partitions.

The System Architecture, the Server Structure and the TPR

The UNS first class service is composed of a number of servers. The clients make requests via the User Agent (UA) which contacts the servers through the UNS Server Interface. In Figure 5.3, the system architecture as well as the server structure is shown. Following the example given above, a name server group $\mathbf{S}_d$ is also illustrated. All the detail of replication and the protocol are transparent to the client. Each server consists of two functional parts: one performs the coordination function, the other performs the participation function.

[5] A smaller replication unit, called *reference* is introduced in Chapter 6.

Every server maintains a Table of Pending Requests in its stable storage besides the data replica. The TPR on a server s contains a $Psid[s]$ which is the identifier of the last synchronisation that s voted, or $-\infty$ if s never voted; a $Nsid[s]$ is the identifier of the last synchronisation in which s promised to participate, or $-\infty$ if s has never promised any and a $PrevR[s]$ which contains the previous requests (which are time-stamped) that the server has voted for.

In addition, a unique identifier *sid* is assigned to each synchronisation when it is attempted by the coordinator. A *sid* can be generate in many ways, e.g. by combining an integer with a server identifier which is unique. For instance, A server with identifier #12 may generate an *sid* (1024, #12). The *sid* is totally ordered, i.e. (1024, #1) > (1023, #2) > (1023, #1). [6]

A sample TPR is shown in Figure 5.4, where the last synchronisation that the server voted for has an *sid* (1023, #1). The current synchronisation that the server is involved has an *sid* (1024, #1). The last request that the server voted for is "123 : modify(UDI, vo, vn)", where 123 is the time-stamp.

5.5.3 The Protocol

A full description of the UFP is given below. Some of the actions which change the states or the data replica are single-site atomic actions. For instance, as a result of receiving a *NextS* message, p will change its TPR if certain conditions are met; this should be done atomically. The coordinator c executes each action sequentially, so does the participant p. The RPC semantics is assumed but error handling is omitted. For example, if c does not get a reply to the message *NextS*, it will not continue to execute **c3**. If a timeout is reported, c may try **c1** and **c2** again.

In the UFP, each client request is associated with a time-stamp issued by a server. A time-stamp used by the UFP is a logical time-stamp, i.e. an integer. A data replica on server s has a version number - $ver[s]$, which equals the time-stamp of the latest update request. Besides the data replica, every server s running the UFP should maintain the following state variables in a stable storage:

- $LastTried[s]$ - the *sid* of the last synchronisation that s tried to begin, or $-\infty$ if s never started any synchronisation. There is an *increase* operation that returns a

[6] Note that the first part of the *sid* is compared first. Only if the result of the comparison is equal, the server identifiers are compared. This will prevent a server from always generating bigger *sid* than another server because its identifier is bigger.

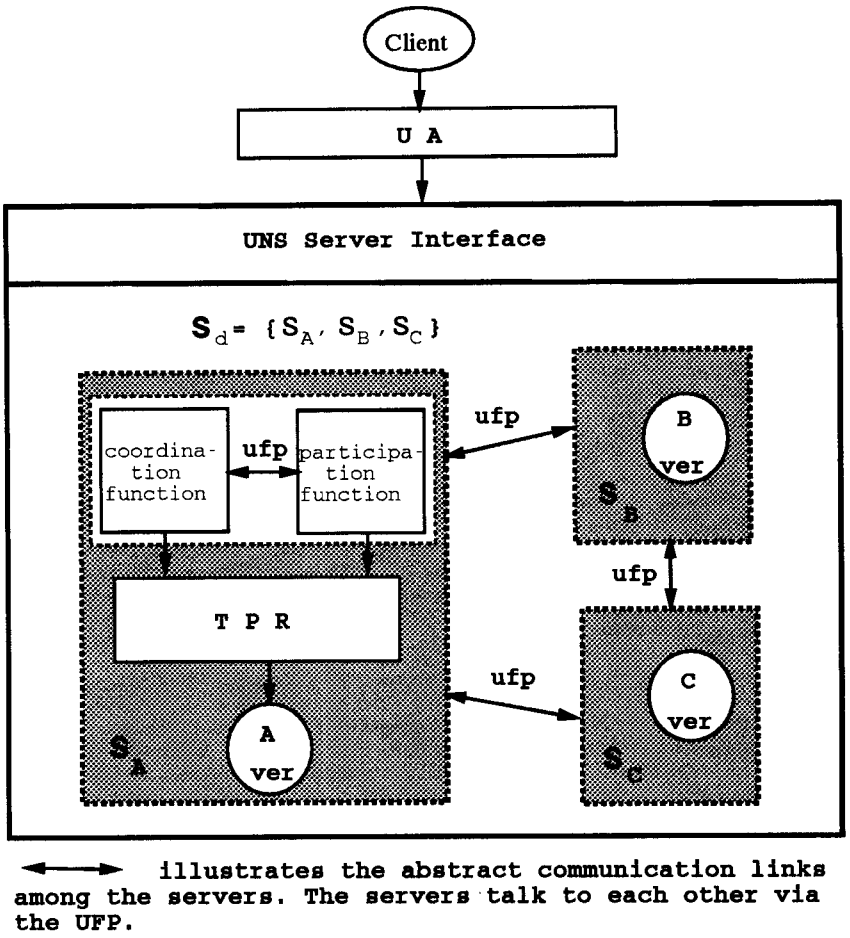


Figure 5.3: The System Architecture

Psid (Previous sid)
Nsid (Next sid)
PrevR (Previous Request)

The structure of the TPR

(1023, #1)	
(1024, #1)	
123	modify(UDI,vo,vn)

A sample TPR

Figure 5.4: Table of Pending Requests

sid greater than the input one.

- $TPR[s]$ - the Table of Pending Requests defined in the last section.

There are also temporal state variables kept or used by s:

- $PrevVotes[s]$ - the set of votes received in $LastVote$ message for the current synchronisation.
- $quorum[s]$ - the set of servers forming the quorum for the current synchronisation.
- $voters[s]$ - the set of quorum members from which s has received $Voted$ message for the current synchronisation.
- $op[s]$ - the request being synchronised, that contains the request in $op[s].op$ and the time-stamp of the request in $op[s].tm$.
- v - the message sent by s on receiving the $LastVoted$ message. v has three parts: $v.op = \{v.op.tm, v.op.op\}$ for the request that s voted last time, $v.sid$ for the sid of the last synchronisation, and $v.vote$ for an OK/REJ message or the current version number of s.
- req - the request being synchronised.
- b, $nsid$ and $psid$ - variables of the sid type.

Phase 1 :

Coordinator c's actions

c1. Try To Start A Synchronisation

$LastTried[c] \leftarrow increase(LastTried[c])$;
$PrevVotes[c] \leftarrow \emptyset$;

c2. Send $NextS$ Message

Send a $NextS(LastTried[c], ver[c])$ to every $p \in \mathbf{S}_d$;

Participant p's actions

p1. Receive $NextS(b, ver)$ Message

if $ver < ver[p]$
then $v.vote \leftarrow ver[p]$;
go to **p2**;
else if $ver > ver[p]$

then p enters recovery mode; exit;
if $b \geq Nsid[p]$
then $Nsid[p] \leftarrow b$;
$v.vote \leftarrow$ OK;
else $v.vote \leftarrow$ REJ;

p2. Send *LastVote* **Message**
$v.op \leftarrow PrevR[p]$; $v.sid \leftarrow Psid[p]$;
Send $LastVote(Nsid[p], v)$ to c;

Coordinator c's action

c3. Receive $LastVote(nsid, v)$ **Message**
if $nsid = LastTried[c]$ and $v.vote = OK$
then $PrevVotes[c] \leftarrow PrevVotes[c] \cup \{v\}$;
else if $v.vote \neq$OK or REJ
enters recovery mode; exit;

Coordinator c's actions

c4. Start Polling Majority Set Q
if $Q \subseteq \{p | v$ from p has been merged into $PrevVotes[c]\}$% Q stands for the quorum%
then $voters[c] \leftarrow \emptyset$;
$req \leftarrow maximum(PrevVotes[c])$;
%the *maximum* operation returns the request having the maximum *sid*.%
if $req.sid \neq -\infty$ and $req.op \neq \emptyset$
then $op[c] \leftarrow req.op$;
else $op[c].op \leftarrow$ any client request;
$op[c].tm \leftarrow increase(ver[c])$;
else exit;

Phase 2 :

Coordinator c's action

c5. Send *BeginS* **Message**
Send $BeginS(LastTried[c], op[c])$ message to every $p \in Q$;

Participant p's actions

p3. Receive $BeginS(b, op)$ **Message**
if $b = Nsid[p]$ and $b > Psid[p]$
then $Psid[p] \leftarrow b$;
$PrevR[p] \leftarrow op$;

p4. Send *Voted* **Message**
if $Psid[p] \neq -\infty$
then Send $Voted(Psid[p], p)$ message to c;

Coordinator c's action

c6. Receive $Voted(psid, p)$ **Message**
if $psid = LastTried[c]$
then $voters[c] \leftarrow voters[c] \cup \{p\}$;

Phase 3 :
Coordinator c's action

c7. Send *Commit* **Message**
if $Q \subseteq voters[c]$
then Send $Commit(op[c])$ message to every $p \in Q$;

Participant p's action

p5. Receive $Commit(op)$ **Message**
Apply $op.op$ on p's replica; $ver[p] \leftarrow op.tm$;
erase the corresponding entries for op from TPR[p];

Now we look at how the UFP works through the three phases.

Phase1:Validation The first phase is the *validation* phase, in which the version number of each replica is checked by exchanging the $NextS(b, ver)$ message between the coordinator and the participants, where *ver* is the coordinator's version number, and b is the unique identifier of the synchronisation to be started. On receiving such a message, a participant compares *ver* with its own. If *ver* is less, it informs the coordinator by a reply message which will cause the coordinator to enter the recovery mode. If *ver* is greater, the participant enters the recovery mode. If the version numbers are equal, the participant sends an *OK* message to the coordinator. When the coordinator gets the quorum needed, it chooses the request to be synchronised. The request is chosen in such a way that if $v.op$ is the request having the biggest

sid, the coordinator must choose it. Otherwise the coordinator is free to choose any client request. The UFP will not enter the second phase if the coordinator can not get enough positive responses or enters the recovery mode.

Phase2:Preparation The second phase is the *preparation* phase, where the chosen client request is sent to every server having voted in the first phase. The coordinator then waits for replies. If a quorum of participants vote for the request, the UFP goes on to the commitment phase. A quorum can not be achieved if there are not enough active servers, or another coordinator had started a higher-numbered synchronisation, or simply because the communication is slow or has failed. In any of those cases the client may get a timeout report. It is up to the client to decide whether to retry when no *ack* has been received from the service. The UFP guarantees that all operations carried out are serialisable, but there is no global ordering for requests. [7] The UFP ensures that every active server processes the requests which it receives in the same relative order.

Phase3:Commitment The third phase is the *commitment* phase, where a request having a quorum is finally made permanent on each replica. A request is committed if and only if it has been recorded by a quorum number of TPRs. In contrast to the Paxon protocol, the message is only sent to members of the quorum by the UFP, otherwise gaps may be left on replicas, making the version number not match the current state.

An Example

In Figure 5.5, a synchronisation **T** is shown phase by phase. The replica set $\mathbf{R}_d$ = { A, B, C }; the NSG or the server set $\mathbf{S}_d$ = { S_A, S_B, S_C }; The number in the figure indicates the order in which the messages are exchanged. Server S_A becomes the coordinator after receiving a client request. S_A Starts a synchronisation **T** by sending out the $NextS(b, ver)$ message. The messages are exchanged as follows:

1. S_A increase its $LastTried[S_A]$ and sends $NextS(LastTried[S_A], ver[S_A])$ to S_B and S_C.

2. Suppose that all the three servers have fresh replicas. On receiving $NextS(b, ver)$, S_B sets $Nsid[S_B]$ to b if b is greater than $Nsid[S_B]$, and then sends a $LastVote(Nsid[S_B], v)$ to S_A. v contains the vote OK and the previous requests etc from the TPR of S_B. Similarly, S_C also sets $Nsid[S_C]$ to b and sends a $LastVote$ message to S_A.

[7] Global ordering is the effort to order the requests by the time they are made.

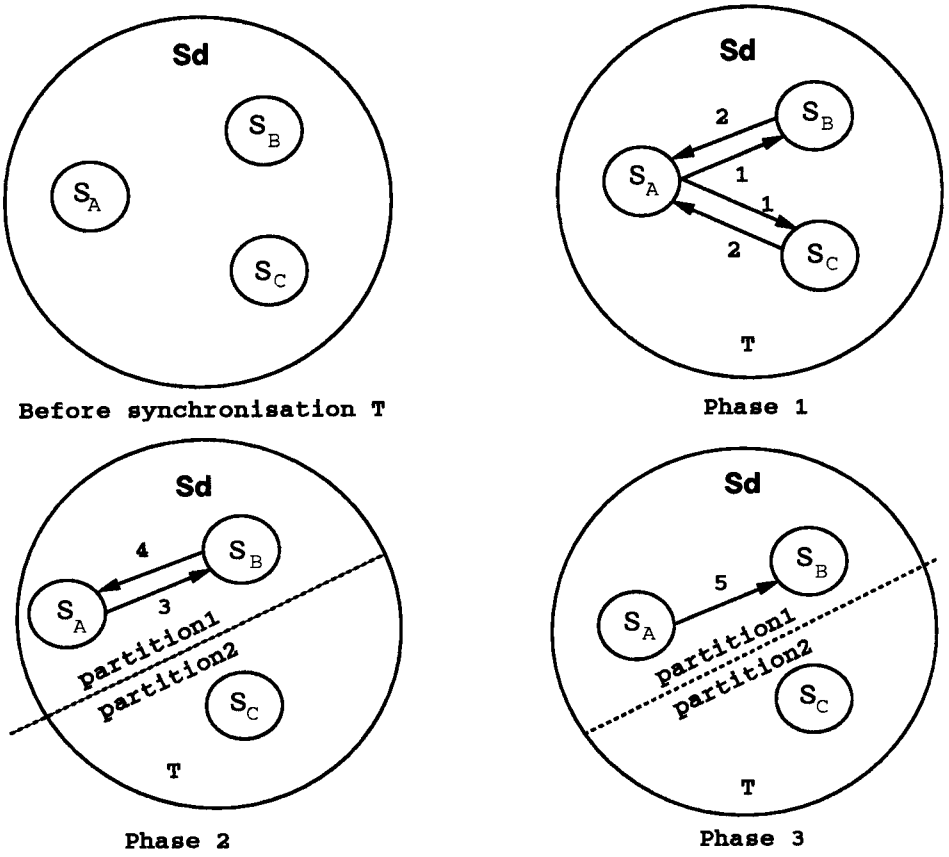


Figure 5.5: An Example of the UFP

3. S_A gets a quorum and chooses a request in the following way. For every v received from S_B and S_C, if $v.sid$ is the biggest and $v.op \neq \emptyset$, chooses $v.op$, otherwise chooses any client request. S_A then sends $BeginS(LastTried[S_A], op[S_A])$ to S_B and S_C. $op[S_A]$ contains the request that S_A has chosen above.

   Note that a failure (partition) now occurs which cuts the communication to/from S_C.

4. On receiving the $BeginS(b, op)$ message, S_B sets $Psid[S_B]$ to b and $PrevR[S_B]$ to op. S_B then sends $Voted(Psid[S_B], S_B)$ to S_A.

5. After getting the $Voted(b, S_B)$, a quorum is again formed by S_A. S_A sends $Commit(op[S_A])$ to S_B for $LastTried[S_A] = b$.

6. The request is carried out by S_A and S_B. The corresponding entries for the request is erased from their TPRs. The client is informed by S_A.

After the synchronisation has finished, the request is made permanent on the replica of S_A and S_B. The version number for S_C's replica is behind due to the partition. In the end of the next section, how S_C will catch up will be explained. Note if a quorum needed can not be formed for a long time, a timeout will occur and S_A will restart the synchronisation.

5.5.4 The Progress Conditions

The progress predicate of the UFP, denoted by **P**, is a conjunction of the condition **P1** - **P5**. The algorithm maintains consistency, but in order to ensure progress, the following conditions must be met:

P1 Operations on servers should be completed within t_{op} seconds

P2 Communications between machines should arrive in t_c seconds

P3 Let **Q** be the set which contains a quorum needed for an operation

P4 At any time there is only one synchronisation in progress

P5 No one rejoins or leaves **Q** within $9(t_{op} + t_c)$ [8]

If a server s, which considers itself the coordinator, initiates a synchronisation request when a majority set of servers are active, then s can expect to pick a client's request within $2(t_{op} + t_c)$ seconds, and to send *Commit* message out within another $2(t_{op} + t_c)$ seconds. If s cannot execute the steps within the time, then either some servers or connections failed after s initiated the synchronisation, or a larger-numbered synchronisation had previously been started by another coordinator. To handle the latter possibility, s has to learn about the larger number in order to start again. In other words, s will be required to restart a synchronisation if and only if: (1) it can not enter the second phase or the third phase of the protocol within a fixed time, or (2) it learns that another coordinator has started a higher-numbered synchronisation. If no server comes back up or goes down when s and a majority set of servers are executing the protocol, an operation will be committed and its results written on every replica in **Q** within $9(t_{op} + t_c)$ seconds.

5.5.5 Elaborations of the UFP

The UFP allows two or more servers to initiate synchronisations simultaneously, since the chance for more than one UNS server concurrently initiating requests against the

[8] The time complexity can be reduced when the UFP is optimised later.

same data is very small. Even if there are a number of synchronisations for the same data at a time, only the one with the highest synchronisation identifier will win. So no confusion results. A timeout will occur if within some fixed time a coordinator can not enter the second or the third phase due to the start of another synchronisation. The Paxon single-president strategy has several drawbacks: first, the president is likely to be overloaded because all client requests have to be passed to him; second, it is less robust in respect of failures; third, it has a less degree of concurrency than the multi-president (multi-coordinator) strategy. Also, a complex election method has to be employed. On the other hand, one drawback with the multi-coordinator strategy is that the progress may be prevented if synchronisations for the same data are started too frequently. However, the frequency of the requests made to the first class service is expected to be low, because updates are far less than queries. Note that having multiple coordinators will not cause inconsistency, since the safety conditions still hold. [9]

As regards the cost of the UFP, a synchronisation involves five rounds of message exchanges and $5N$ messages if there are N participants. With the single-president method, after a president is elected, the protocol has three message delays and $3N$ messages. The communication cost can be further reduced if a *commit* message is sent together with the *BeginS* message of the next synchronisation, i.e. only $2N$ messages are exchanged, and the time complexity is 2 rounds of messages. However, getting a president elected can take a long time and great care must be taken when the current president stops functioning and a new president has to be chosen [Mann 89]. There are three reasons that the multi-coordinator method is used by the UNS: firstly, the UNS can afford the cost of having multiple coordinators since the frequency of requests is expected to be low; secondly, it is easier to implement since no complex algorithm is needed to get a president elected from time to time; thirdly, client requests to the UNS are mostly single-shot transactions.

After a request has been committed, it is not necessary to go through a synchronisation for it again. After the commitment, the corresponding TPR entry for the request can be replaced by the forthcoming requests, so that the TPR will not get arbitrarily long. However, the information in the TPR can not be thrown away before the request has been committed. Otherwise inconsistency may be caused. For instance, suppose that less than a majority of replicas have recorded a committed request numbered 123, and some TPR entries for the request are erased. Later a new coordinator got a quorum, which did not include any one knowing the request 123, and got a different request also numbered 123 committed. When the two groups of servers meet together, inconsistency will be found in the replicas. In order to avoid the problem, every server keeps the TPR entry until the

[9] See [Lamport 89] for proof of consistency.

request has been recorded on its replica. The predicate **I** below is therefore always true.

TPR(s) denotes a set of the current states kept by a server s, such that

$$\begin{aligned}\text{TPR}(s)= \{e \mid\ & e \textit{ contains the states of a pending request} \\ & \vee\ e \textit{ contains the states of a committed request} \\ & \vee\ e \textit{ is } \emptyset\ \}.\end{aligned}$$

Let **Q** be the quorum needed to carry out a synchronisation. A replica r stored by a server s is defined as a data set: $\mathbf{D}_r(s) = \{u|\ u \textit{ is a committed request}\}$

Predicate **I** is left true after a synchronisation.

I : A request u is committed $\Longrightarrow$ $\{\ \forall s\ (s \in \mathbf{Q} \wedge ((\{u\} \cap \mathbf{D}_r(s)\ \neq\ \emptyset)\ \vee\ (\{u\} \cap \text{TPR}(s) \neq \emptyset)))\}$

Proof:
Suppose u is committed; s is a member of **Q** which voted for u during the synchronisation. u is reflected by TPR(s) only after a *BeginS*(sid, u) message is received by s during the 2nd phase of the UFP. During the 3rd phase of the protocol, on receiving a *Commit* message, s adds u to $\mathbf{D}_r(s)$, and then erases u from TPR(s); the above two actions are made atomic. The coordinator only sends *Commit* to those belong to **Q** during the 3rd phase of the UFP. Therefore, if there was a s in **Q** such that: u was neither recorded by TPR(s) nor written on $\mathbf{D}_r(s)$, then s did not see either *BeginS*(sid, u) or *Commit*, that is contradictory to the assumptions that s is in **Q** and u is committed. Thus **I** is left true after the synchronisation. □

Since I is always true, a coordinator will be able to learn any committed request from a server having voted for it. Following up the example given in section 5.5.3, the server S_C failed during the second UFP phase. Suppose S_C resumes communication with the other two servers and receives a client request. S_C begins a synchronisation and will enter the recovery mode during the first phase because its version number is behind. S_C can learn the missing requests from either S_A or S_B (see also next section for recovery from failures).

Finally, a fine-grain replication can be used to allow higher concurrency. For instance, different requests to different entries in a directory may be accepted at the same time, which is impossible if the directory has only one version number. If two updates to the same entry are issued concurrently, the one with a larger time-stamp always wins.

5.5.6 Discussions

Local Atomic Action

Although no multi-site transaction (atomic action [Lampson 81]) support is needed by the UFP, single-site (local) atomic action must be used. An action can be described by operations taken by a server and the results of them. For example, a participant's action upon receiving a *BeginS* message includes receiving the message, comparing variables, and setting variables. The details of implementing a local atomic action mechanism are outside the range of this research.

Resolution of Conflicts

Despite the hierarchical naming used by the UNS, it is possible for conflict names to exist. For example, here are two requests received by a server:

1234: Parsley is a laser printer
1236: Parsley is a line printer

Users would get confused if the above two requests are both accepted by a local name server. To avoid getting conflicts, the following observation is useful: if request u and u' were committed, sid was associated with u, sid' with u', and $sid > sid'$, then the coordinator must have learned u' before it chose u.[10] A coordinator can find out a committed request either from a TPR or a data replica, since **I** is always true. Therefore, for the addition operation, if the name has already existed, the coordinator can find it out and will refuse any request to add the same name again.

Any conflict will be detected during the first phase of the protocol. For instance, the coordinator for request 1236 sends a $NextS(sid, 1236)$ to every other server in the NSG. If an earlier request with time-stamp 1234 is committed, the coordinator should be able to learn it (because request 1234 and 1236 share the same key, i.e. "Parsley", in the add operation), or the pending request 1234 from at least one participant. If the coordinator has neither leant about request 1234 nor its outcome, it must have an older version of data, or request 1234 has never been committed. In the later case, request 1236 can be picked up for synchronisation. In the former case, the coordinator enters the recovery mode for catching up.

[10]The decree-ordering property is described in [Lamport 89].

For update operations, however, it always makes sense to apply an update to the UNS database. A value of a data entry is initially empty and then determined by a sequence of update operations. The set of updates that have been applied defines the state uniquely.

Read Only Operations

In the UFP, a **c** option of query (slow-read) is designated to get the most recent data. Three methods may be used to implement such a query. In the first thought, a query with the **c** option means to go through a synchronisation. Alternatively, a time-stamp can be associated with the query. Given the time-stamp, a server can decide whether the data is more recent or not. The third method is to appoint a specialist-server as the authority to answer any enquiries made to a particular portion of data. None of these methods seem satisfactory in regard to the naming semantics. It is very expensive to go through a synchronisation each time an enquiry is made. To ask the client specifying time-stamps when making an enquiry does not seem suitable either because the client usually does not remember the time-stamps. The specialist-server strategy is also not very interesting because it is in fact a primary-copy method, which is less robust than the quorum consensus methods.

It is possible to learn the most recent data by reading a quorum, as seen in many quorum consensus protocols such as weighted voting [Gifford 79]. Reading a quorum requires only two round of message exchanges. Although the Paxon protocol is quorum consensus, it is tricky to read a quorum because there is no guarantee that the current data can be found. [11] This can be seen as one of the major difference of the Paxon protocol from previous quorum consensus methods.

As mentioned earlier, the UFP uses quorum for fault tolerance and the flexibility of adjusting the votes for read or write among the replicas. The UFP is developed to get the current data through a quorum read. The pseudo code for this is shown in Figure 5.6, where the request u is defined as pending for it appears on less than a quorum number of TPRs, i.e. it has yet to be committed. This could be a result of a partition which prevented the synchronisation from completing the second phase. If a server s receives an enquiry at the moment, and becomes the coordinator, s may learn about u or not in the end of the first phase, because u appears on less than the quorum number of TPRs. After the first round of message exchange, if s has the current version number and a quorum, it checks the votes received. If s learns about u and its version number is smaller than u's

[11] In the Paxon protocol, it is possible that a passed decree was only reflected on one or two ledgers but not all of them in the quorum.

```
// Coordinator's code for slow read
char* query(UID key, char opt = c)
{ // in phase:1
    ... // The server sends NextS to the participants and waits for replies
    if (ver is current && there is a quorum) // ver contains the version number
        if (u is chosen from the participant's votes)
            if (key == u.key && u.tm > ver )
            // u.key, u.tm extracts the key or time-stamp item respectively from u
                enter phase:2;
            else return(_lookup(key));
        else
            return(_lookup(key));
    else enter recovery mode or re-try;
// in phase:2 - Phase:2 omitted
}
```

Figure 5.6: Slow Read Operation

time-stamp, and also the enquiry concerns u, u should be synchronised before s answers the enquiry. If the enquiry does not concern u, the enquiry is answered by s through a local lookup, i.e. _lookup(). If the query concerns u but u has smaller time-stamp, u is ignored (u will be erased from the TPR later) and the enquiry is answered by s. If s is out of date, it enters the recovery mode. If there is not a quorum during the first phase, s can re-try the request later.

In summary, a query with the **c** option will always return the most recent data. It does not enter the second and the third phases unless a pending request with a time-stamp larger than the current version number is found. Figure 5.6 shows the code of the slow-read operation with one pending request. If there are more pending requests, the algorithm is very much the same except more comparisons are needed.

In addition, most enquiries are made to the secondary copies as discussed later in this chapter, so the number of enquiries to the first class service is much less than that a traditional name server would expect. Usually, the first class service will call back the secondary one to propagate the recent updates made. In case this fails, a secondary server may consider its copy suspicious, in which case it makes a query with option **c** to the first class service in order to refresh its own copy.

Add Operation

The add operation is more complex in comparison with the others because it needs validation. As discussed earlier in this section, the UFP has the property for sorting out conflicts. The first phase for the add operation can be implemented in the same way as the slow-read operation, except that it checks with the local lookup operation whether the name to be added exists or not. Once the validation is done, the rest of implementing the add operation has no more difference from that of other operations. If the name already exists, the error *NameAlreadyExist* is reported and the process is aborted. If there is an add operation started earlier, which is pending and conflicts with the current one, the UFP rejects the current request. However, it is possible that a client making a request later gets its name added while the earlier one is aborted, because there were not a quorum for it and it is not learned by the current coordinator. The UFP only guarantees that no conflict exists, but does not guarantee to process them in the global order.

Recovery From Failures

As mentioned before, a UNS first class server runs in one of the two modes: *normal mode* or *recovery mode*. After receiving a *NextS* message from the coordinator, a participant can decide independently whether its replica is stale or not. If the participant's version number is smaller, it enters the recovery mode. If the participant has a bigger version number, it will send a reply message containing its version number. The message will cause the coordinator to enter the recovery mode. The major purpose of the recovery mode is for a server which is just back from a crash or a partition to be able to catch up with the current version. The methods for recovery are well developed, such as the following:

- full-copy
  In recovery, a full replica is copied to replace the old one. The full-copy recovery is simple to implement, but if the size of a replica is too big, it will be slow, especially when remote data transfer is involved.

- logging and check-pointing
  In regard to this sort of method, a log is maintained by each replica in a stable storage, which contains the recent updates that have been committed. On recovery, only the log is replayed rather than the full copy, so it is more efficient than the full-copy method. The log should be kept in a way that allows the updates to be applied to the obsolete replica. Check-pointing must be employed since a log can not be arbitrarily long. This dissertation does not address this kind of method.

- partial copy
  This method copies only the part of data that is different from the current version. If fine-granularity is used and time-stamp is associated with each data unit, the method is better than the full-copy provided that the comparison is fast.

Recovery can be carried out in background and only after it finishes, a server can run in the normal mode. The current version of data replica can be found by a slow read.

Maintaining the Replica Set

Every member in the replica set of a directory must know how to contact the others. This can be done by storing a list of server names on each replica. A replica set is subject to change. For instance, a replica is added or removed to or from a replica set. In Chapter 6, how to maintain information for a replica set will be discussed.

Properties of the Protocol

In summary, the UFP protocol has the following properties:

- optimistic concurrency control can be used;
  Updates are always allowed except for addition. In case a new name is to be added, possible conflicts should be resolved by the UFP so that only one of the names is accepted.
- granularity of concurrent control may be chosen;
  The unit of replication control may be a whole directory or a portion of data within a directory.
- no underlying distributed transaction support is needed;
- the cost is less expensive than that of a three-phase commitment protocol, but has similar properties, e.g. fault-tolerance is kept;
- most reads are less expensive;
- quorum consensus can be combined with the algorithm;
- one copy serialisability is maintained.

5.5.7 Related Work

In [Ladin 90], three kinds of ordering are provided: *client-ordering, server-ordering and global-ordering.* With the client-ordering, updates and the propagation of their effects are done asynchronisely. The server-ordering ensures that the update operations are carried out in the same order at all servers, and the global-ordering guarantees "immediate" installation of the update operations with respect to all other operations. The UNS can not take advantage of the client-ordering for the update operations, because it is very difficult to make *gossip* schemes reliable for large and heavily replicated naming systems. The server-ordering and the global-ordering provide the same guarantee as the UFP, but slow down queries and add more complexity to the system, since the former requires an extra view change method [El-Abbadi 85, El-Abbadi 86], and later uses the three-phase commitment. When in process, the global-ordering also blocks all other operations. The goal of the UNS is different however, it does not prevent clients from seeing temporary inconsistency. It is up to the client to decide whether a fast-read is acceptable or not. If a client does have a bad experience because of using inaccurate data, the UFP can help by means of a slow-read operation.

In contrast with the UFP, other voting schemes such as the one used by some file systems [Gifford 79, Bloch 82] rely on an underlying atomic action mechanism, which adds considerable overhead to the consistency control protocol and also reduces efficiency.

The UFP has some advantages over the well-known commit protocols, namely the two-phase and the three-phase commitment. The two-phase commitment is less fault-tolerant and less robust than the UFP, for instance, it occasionally blocks. The three-phase commitment is free from blocking, but more expensive and less flexible than the UFP. For each transaction, there are 5 rounds message exchanges with the three-phase commitment. After optimisation, the UFP can be fulfilled in three message delays. Thus the UFP is less expensive and equally robust in respect to the three-phase commitment. Elaborations are made to the UFP to obtain more efficiency and robustness.

In the UFP, updates are only allowed at the first class service. However, availability and performance of lookup operations are improved by introducing the second class servers, which keep read-only copies of the data stored by the corresponding first class servers. When there are no failures, the load of the first class servers is mainly from update requests in addition to enquiries from special applications. When there are failures, such that some updates are not seen by some of the first class or the second class servers, a *slow read* to the first class service is required to get the accurate naming data. A UFP *slow read* will return to the client the most current data in question. When the first class service

can not be accessed by clients due to failures (network partitions or server crashes), no more updates can be carried out by the service and the slow read also becomes impossible. However, it is possible that some updates made successfully just before the cut-off have yet to be reflected on the secondary copies which cannot contact the first class service, in this case slow reads are delayed. Soon after the partitions rejoin each other, or the first class service comes back up, a slow read or update can be carried out again and the result is spread to the second class servers.

Client enquiries are not directed to the first class service except when specified. Temporary inconsistency between the first class service and the second class service will not be noticed by most of the clients. However a few clients may notice the inconsistency if a slow read is not enforced. For example, the client who initiated the update makes a follow-up enquiry, which is received by a secondary server that has not been informed about the update. This seems acceptable under most naming semantics. Should different applications be considered, a suitable query strategy can always be chosen.

5.6 The Protocol For Update Propagation

Experience with Grapevine has shown that a relaxed form of consistency is very useful in practice [Birrell 82, Schroeder 84] although update propagation delays are sometimes surprising. As mentioned in previous sections, when a naming database is replicated at many sites and the number of sites grows, maintaining mutual consistency for update operations becomes a significant problem. A two-class name service infrastructure is introduced to tackle the problem by maintaining strict consistency among the first class servers, and loose consistency among the secondary servers. The protocol for the first class servers has been developed in Section 5.5. This section mainly concerns the update propagation mechanism to be used by the secondary servers. Several techniques have been developed such as the *sweep* algorithm [Lampson 86], lazy replication [Ladin 90] and the *epidemic* algorithms [Demers 87]. This section explains that the asynchronous algorithms are compatible with the two-class name service infrastructure. Some implementation issues are also discussed.

5.6.1 Using Asynchronous Methods

The replication control problem encountered by the UNS is complicated by the requirement of large scale, high availability and autonomy. It is very difficult for any single replication control mechanism, either synchronous or asynchronous, to meet the demands.

The problem with the synchronous methods is that it is extremely difficult to implement a tight-coupled replication mechanism which scales well and allows autonomy. On the other hand, none of the asynchronous methods reviewed in Section 5.2 has the ability to resolve the inconsistency caused by failures. The two-class name service infrastructure is therefore introduced to combine different replication control mechanisms. As analysed in Section 5.2, the asynchronous replication control methods are suitable to applications that require high performance, high availability and autonomy. The three methods seen in Section 5.2 have been implemented by large name services such as the DECdns [Martin 89], the Clearinghouse at Xerox [Demers 87], and a number of services at MIT [Ladin 88, Hwang 88]. Both *sweep* and *gossip* require that a replica know all other replicas. It is almost impossible to get complete knowledge of others in a vast, distributed and mistrusting computing environment. The experience with the Clearinghouse algorithm shows that random propagation converges fast while causing less traffic on the communication network and requiring few extra data structures [Demers 87]. The epidemic algorithms are the least expensive and easiest to implement of the three. Previous mechanisms for maintaining replicated database depend on various guarantees from the underlying communication protocols or maintaining consistent distributed control structures. For instance, the QUIPU directory service relies on a reliable multicast support for propagating updates [Kille 89]. In Lampson's sweep, the pointers must form a ring. The epidemic algorithms seem the best choice for the UNS second class service.

When a secondary copy is created, it is registered with a first class server so that updates can be propagated to it. Each first class server can look after a number of secondary copies; the first class server and the corresponding second class servers form a server group called a *cluster*. A replicated directory object consists of several clusters. The first class servers call back their clusters to push updates ; no reliable message delivery is required because only loose consistency has to be achieved. It is possible that during the propagation of updates, some failures occur so that some secondary copies are missed. A randomised anti-entropy can be used among secondary copies for detecting inconsistency and converging.

5.6.2 A Few Implementation Issues

Version Numbers

In the last section, the compatibility of the epidemic algorithms with a synchronous replication control method has been investigated. To implement the update propagation algorithm, each copy should be tagged with a version number which helps to detect inconsis-

tency and indicates the necessity of accessing the first class service for accurate data. For instance, clients can learn how recent the data is if given a piece of data with a time-stamp. The version number is advanced when the first class service has carried out the updates successfully.

Push vs. Pull

In the UNS, *push* means the firsts call back the seconds if there is any update made recently; *pull* means the seconds contact the firsts periodically to refresh their read-only copies. Both methods have advantages and disadvantages. If the data change rate is low, pulling indiscriminatingly from the first class servers by the second class ones is likely to generate heavy traffic on the communication links. Call-back is used by the firsts instead to push updates to the secondary servers. There seems no way to prevent some copies from not seeing the updates sent for some reasons, namely site or communication failures. What the firsts can do is to keep trying in the case of exceptions from RPC calls. However, handling failure of call-back in a large distributed system could occasionally be well beyond the firsts' ability. Alternatively, the firsts can remember recent updates for a while and combine those updates into the same message containing the new updates, expecting those which missed something before to receive them. No matter which method is used, there is no guarantee that every copy sees all updates sent by the firsts. Secondary servers which come back up from failures may pull the firsts to learn the recent updates. Secondary servers can also run anti-entropy among themselves rather than rely on the firsts.

The Number of Sites

For a global context to be replicated widely around the world, a cluster can become too big to handle. One solution to this problem is to reduce the number of secondary servers which are registered with the corresponding first class servers, and to allow the epidemic algorithms to be run among a group of secondary servers. For instance, an organisation having hundreds of sites may register only several selected sites with the first class servers, and those sites then send the updates to some nearby sites by rumour mongering. Similarly, the anti-entropy should be run for spreading the updates to the few sites that do not receive them as a rumour.

Choosing Partners

The way of choosing partners when issuing rumours or doing anti-entropy can affect the network traffic significantly. Experience with the Clearinghouse [Demers 87] suggests that choosing partners uniformly results in a higher network traffic than spatial distribution of rumours. By spatial distribution we mean that the partners are chosen according to their distance to the "infective" site on the network. An infective site is a site willing to share rumours with the other sites. Analyses and simulations on the Xerox CIN reveal that spatial distributions converge nearly as rapidly as the uniform distribution.

5.7 Caching in the UNS

In this section, caching is discussed in more details. At first, previous work is reviewed; then caching in the UNS is discussed; finally comparison with the previous work is performed.

5.7.1 A Brief Review of Caching in Naming Systems

In general, a cache is useful for improving the performance of computing systems. For instance, in a distributed system more expensive accesses to a remote server can be avoided by caching the data nearby. Most file systems use caching for good performance. Much work has been done on caching in naming systems [Terry 87, Birrell 82, Mockapetri88, Lampson 86]. In Grapevine, the strategy of treating cached information as hints works because people change their mailboxes infrequently and recovery from use of stale data is possible. Similarly CSNET [Solomon 82] also used caching for mailbox information. The DNS employs a time-to-live caching approach. Each database entry has a time-to-live field whose value can be obtained at the time the data is registered. However, there is not much suggestion of how the value of time-to-live should be decided. In contrast, [Terry 87] allows more control of cache accuracy by the cache manager. Given the current age of the data and its expected lifetime, the cache accuracy can be estimated to allow the cache manager to maintain a desirable accuracy level. The approach tries to guarantee a performance benefit from more accurate caching instead of simply increasing cache hit ratio, so that clients will not waste time to use bad cache data. The way of using cached hints in distributed systems is shown in Figure 5.7. In [Mann 87], a client-based name prefix caching approach is taken for performance-critical objects such as files, windows and executing processes in distributed systems. The approach allows clients to send their

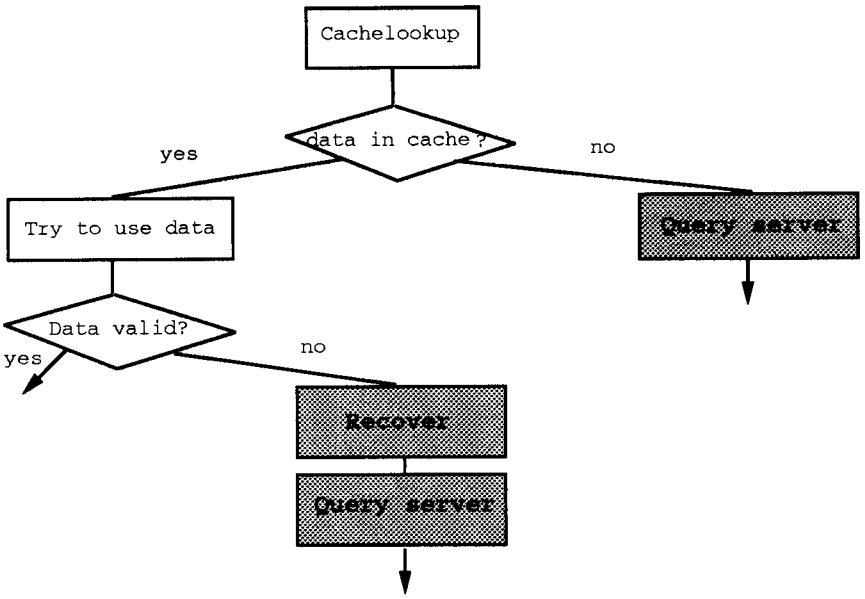


Figure 5.7: The basic algorithm for using cached hints

name-mapping request directly, in the common case, to the manager that maintains the object. If a client cache misses, multicast-lookup can be used among a set of managers that might cover the name.

Problems with the cache model in Figure 5.7 are:

- Whether use of the obsolete data can be detected and how long it takes. The experience with the CSNET [Solomon 82] name server suggests that the time to detect bad cache information may vary from seconds to days. It is undesirable for some clients to experience a high lookup cost in addition to recovery cost caused by using poor accuracy cached data.
- Whether recovery at the application level is acceptable. If it is less desirable or even impossible for some applications to use hints, other methods to maintain data integrity should be employed.
- Which server the client should contact for validating the cached data. In Figure 5.7, "query server" means to lookup a cached data which appears not to be working. In systems like Grapevine and CSNET, it is not possible to find a server with the most recent data unless all replicas have been contacted. On the other hand, clients may not know which particular server to contact since it is made transparent by the service.

In summary, [Terry 87] focuses on maintaining a desired accuracy level of cached informa-

tion while many others concentrate on hit ratio only. [Mann 87] allows good performance within the scope of a local administration or among several object managers, but a global name service or a more powerful multicast mechanism is needed to deal with cache misses. Considering a large distributed naming system with heavy replication, if a piece of cached information appears not to be working, it can be very difficult to decide which server should be contacted to validate the information. It is the common case that one can not decide whether a cached data is valid or not until one has tried it out at all possible replicas. Many applications cache hints because higher level recovery from using wrong data is allowed. However, application level recovery has its cost and needs efforts from outside the system. It is not clear whether all naming applications can use hints satisfactorily. Instead of making efforts to improve hit ratio or estimate the accuracy level of cached information, the UNS implements the two-class name service infrastructure while using caching as well. Invalid caches are discarded when they are first found not working; clients or servers can then contact the UNS by indicating the preference of queries. For applications which can not afford higher level recovery from inconsistency, a query with a specified requirement of accuracy can be applied and hints are not used.

5.7.2 Caching in the UNS

Name lookup in a large system such as the UNS is likely to be expensive, especially when a directory is stored far away. It is desirable for a client or a server to be able to cache the result of a name lookup which may be used again in the near future. A UNS server caches its own name and names of the directories it stores so that it does not rely on the UNS to lookup its own name. There are three kinds of data in the UNS, *replica*, *read-only copy* and *cache*. The major difference between a client cache and a read-only copy, which is also called an *official copy*, is that the latter is automatically refreshed by the system and algorithms are used to maintain its accuracy, while the former is not. Read-only copies contain a whole directory but a cache can be an entry or a directory or whatever a client may like. A cached data is guaranteed to be up to date to the time indicated by a time-stamp associated with it. It is up to the client to decide whether such a guarantee is acceptable or not. For example, a mailbox name for a university student may be valid until 30th September 1991. The client can always obtain the most current data by indicating the preference of a query operation.

The reasons that a piece of cached naming data fails to work could be: that a recent update has not been seen since it is impractical for the service to keep track of clients of caching, the server having the corresponding information is out of accesses, or just because the

communication network is slow. In a case that a server is out of contact, there should be a server or connection failure report rather than a rejection when using the data. Sometimes it is not easy to tell a failure from a slow response, a timeout report is got by the client rather than an error message. It is for the client to decide whether to contact the UNS or just try the cached data again later on. The implementation of the two-class name service infrastructure enables the client to find out what goes wrong in case of need.

Compared with previous approaches, the UNS needs little effort from the client of caching to keep the cached data accurate. Client can always get a satisfactory answer either from the service or the cache. There is also little restriction to the scope in which the mechanism works. The UNS does not rely on underlying multicast or other such support from the communication network. It is worth mentioning that although the UNS validates caches on use as in the previous approaches, the cost of doing so is less expensive since naming data is more properly replicated. For instance, a local directory may not be widely replicated so that validation of a cache with the server is not expensive; there may be a secondary service available with good enough data to save the client from going to the first class service. However, more experiment in a real world is needed to give a more precise judgement.

5.8 Summary

In the design and implementation of large distributed name services, replication control plays a very important role. The benefits from replication are obvious in terms of performance and availability. However, if correctness is considered, it proves that a tradeoff always exists between correctness and the others. Previous work in this area tends to engage with a single replication control method under all circumstances, although some of them have taken semantics into account; the resulting systems either suffer inconsistency for availability or performance gains, or do not scale properly. One contribution of this research is to investigate many existing naming systems to get a clearer idea of how name services are partitioned and distributed, how naming data is used and how replication should be done. The two-class name service infrastructure is then introduced to allow combination of a tight-replication control protocol with a loose one. The UFP is studied and implemented as an example of the two-class name service infrastructure. The epidemic algorithms and caching are also explored. In conclusion, the UNS allows reads to proceed at anytime except when specified otherwise or when no servers are available. The UNS permits most updates to happen although a few of them may be blocked until failures are repaired. The *official copies* on the second class servers are refreshed by the

first class servers so that any inconsistency can be resolved eventually. Finally, caching by servers or clients can improve performance substantially.

Chapter 6

Dynamic Service Configuration

6.1 Introduction

In this chapter the configuration of the UNS is investigated. Two aspects are considered: one is how to keep the service configuration data accurate, which determines reliable navigation as described in Chapter 4; the other is how to manipulate replication sets. This research focuses on dynamic service configuration. The effects of configuration changes on navigation are also explained.

Previous design work on naming systems pays little attention to dynamic service configuration. Dynamic configuration is essential to support the reconfiguration of a name service as it continuously evolves. One design requirement of the UNS is a long service life. During the life time of a name service, many changes may be made for either organisational or operational reasons; many of these could not be foreseen in detail when the name service was designed. Dynamic configuration allows a name service to adjust itself to reflect changes, such as adding a new server to offload an existing server. It should be carried out without modifying functional components or stopping the entire system. Fault isolation is also required.

The rest of this chapter is arranged as follows. General aspects of name service configuration and related work are given in Section 6.2. In Section 6.3, the UNS dynamic service configuration is discussed. The algorithm and the correctness proof are also presented. Finally, Section 6.4 concludes this chapter.

6.2 On Name Service Configuration

In this section, the general aspects of configuring name service are studied. The requirements for establishing the service configuration are outlined, and the existing mechanisms are examined.

6.2.1 Fundamentals

In a distributed computing environment, a number of services such as the time service, the name service, the mail service and so on are run jointly by a number of organisations. In order to provide clients with a uniform set of services, cooperation is generally required. Unlike a centralised system, a distributed system comprises several servers at different sites. Considering a global name service in particular, hundreds or even many thousands of name servers, each of which is responsible for one or more portions of some name space, are scattered widely around the world. Name service configuration concerns various aspects such as follows:

- how many servers to have initially
- what information to be stored at each server or user agent for a start of name resolution
- what information to be stored at a server for passing client requests to other servers
- when to add or remove a server
- where to set up new servers

There are more aspects to be considered if replication is used:

- how many replicas to have
- where to put a replica
- how to add or remove a replica

It is required that individual servers work cooperatively to form a large scale name service. Various approaches in constructing services lead to different complexity of service maintenance.

Different Approaches

A distributed name service consists of a number of name servers. For convenience, those storing inter-domain contexts are defined as *global* name servers; those responsible for resolving names maintained by local administrations are *local* name servers.

There are five kinds of configurations found in previous naming systems. The first is the *Direct Access* approach [Schwartz 87]. In this approach, the interfaces to local name servers, which are usually heterogeneous, can be obtained by contacting an overall name server - the global name server, which handles all these interfaces. Clients can then access a local name server directly with the interface. The global name server has one level mapping from global names to addresses of local naming authorities. Clients are well aware of the difference among local name servers. Access and location transparency are not offered. Transparency can be supported by employing a so called *Naming Semantics Manager* introduced in [Schwartz 87].

The second kind of configuration is called *Re-registration* [Schwartz 87]. Any name to be shared globally is re-registered in the global name server so that clients can look up it via the global server. Local servers can not update their own portions of data which are shared by the global server without coordination, otherwise consistency between the global server and local servers becomes difficult to handle.

The third kind of configuration is *Chaining* [Linden 90]. It is similar to Re-registration but the global server does only a single level mapping between the client and the local server. Unlike the Direct Access Approach, any access to a local name server has to go through the global name server. The global server masks differences of local servers. However, the global server can become a bottle-neck since every request has to go through it.

The fourth kind of configuration is *Federation* [Linden 90]. In principle, the global name server is distributed over all the participating servers of the federation, and acts as the front end of them. It is a more general approach than Chaining. There are a set of protocols that must be adhered to by every server in the federation. The ANSA Trading Service uses this approach [Linden 90].

The last kind of configuration is *Standardisation* [X.500 88]. There is a uniform way to organise name servers. The client contacts a Directory User Agent to start with, which further connects to one or more Directory System Agents (DSAs) until the client request is carried out. It is not necessary to distinguish a global server or a local one, since both of these are DSA. In contrast to the Federation approach, there is only one Directory Access Protocol that every DSA follows.

Dynamic Configuration

Configuration may be *static* or *dynamic*. By "static" we mean that the service configuration can not be changed while the service is running. By "dynamic" we mean that the service configuration may be changed while the service is running, without stopping the entire system. Since it is neither possible nor desirable to stop the whole service, which is jointly run by many name servers belonging to autonomous organisations, the latter should be used by the UNS. Dynamic configuration allows a name service to be reconfigured when it is running, while most of its clients can still make operation requests as usual. Ideally, clients of a name service should not perceive that the service configuration is currently being updated. However, in real life the service may perhaps be seen running slowly as a result. It should be noticed that in some large naming system designs, for instance the DNS, configuration is implicitly assumed to be very stable, so that means for tackling dynamic configuration is not provided as an integrated mechanism. Configuration is either done in an ad hoc manner or left to local administration so that there is no guarantee of correctness. The UNS puts dynamic service configuration into its system architecture for robustness.

Configuration Data

As it has been indicated in Section 6.1, configuration may include two kinds of data: one is responsible for navigation; the other is for replication. Data for navigation is essential for operating the naming system correctly. Data for replication is not only for running the replication control protocols but also for making navigation fault-tolerant.

Obtaining the Initial Configuration

Upon receiving client requests, a user agent or a global server contacts a local name server with some cached information in hand. A user agent usually caches the addresses of a number of servers in order to contact them. It depends on some lower level locating primitives to obtain the cache initially. For instance, a broadcast mechanism on a local Ethernet.

6.2.2 Requirements and Related Work

As discussed in the last section, configuration plays an important role in maintaining the name service navigation and replication sets. This section examines requirements to allow configuration of the UNS. The following three cases are considered the most important.

1. Keeping navigation functioning properly
2. Changing the replication information safely

3. The algorithms for dynamic configuration

Related Work

In Section 6.2.1, five kinds of name service configurations were described. The UNS falls into the standardisation catalogue. In implementing this kind of service configuration, two markedly different approaches are outlined below: The *reference approach* found in [X.500 88], and the *replication approach* used in the GNS [Lampson 86]. Replication is not an integrated mechanism in X.500. By design, X.500 can live without much replication, though it allows replication for robustness. In contrast, replication is integrated to the GNS. The GNS provides operations for manipulating replicas, and sets server invariants which require replicated data to enable navigation. Besides the fundamental difference of design, there are further distinctions between the two approaches.

In X.500, naming data is maintained by the Directory System Agents (DSAs), i.e. name servers. Each DSA contains two types of information: directory information and knowledge information. The server navigation is done by using knowledge information (called knowledge references). A X.500 Subordinate Reference (SR) is used by the DSAs for walking down the Directory Information Tree (DIT). A SR contains both names and addresses of the DSAs holding the directory information, i.e. the naming data in question. A DSA also keeps a Superior Reference (SR) (i.e. names and addresses of some other DSAs closer to the root) for walking up the DIT. Additional spaces and methods are required for maintaining the knowledge references.

The three server invariants set by the GNS have been discussed in Section 4.2.4. The first two of them ensure navigation moving downwards the naming network and no loop is formed; the last one ensures the root is reachable. An example taken from [Lampson86] (reproduced in Figure 6.1) shows how these invariants are maintained by the service configuration. In the figure, an example directory hierarchy is represented by ovals and thin lines; an example service configuration is represented by rectangles, thick lines and dashed lines. The server "ANSI/Mass" keeps a replica of the directory DEC and PRIME for walking down the naming network; the name server "ANSI/DEC/alpha", "ANSI/Prime/1" and "ANSI/Prime/2" store the directory ANSI for moving up to the root server "ANSI/MASS".

There are two problems with the above strategy. In the former case, if ANSI has many subordinate directories rather than just DEC and PRIME, it is expensive for the server "ANSI/Mass" to store the replicas for all ANSI's children. In the later case, if the root directory ANSI is kept by all servers responsible for ANSI's subordinate directories, consistency is difficult to maintain. As mentioned in Chapter 5, the *Sweep* algorithm requires

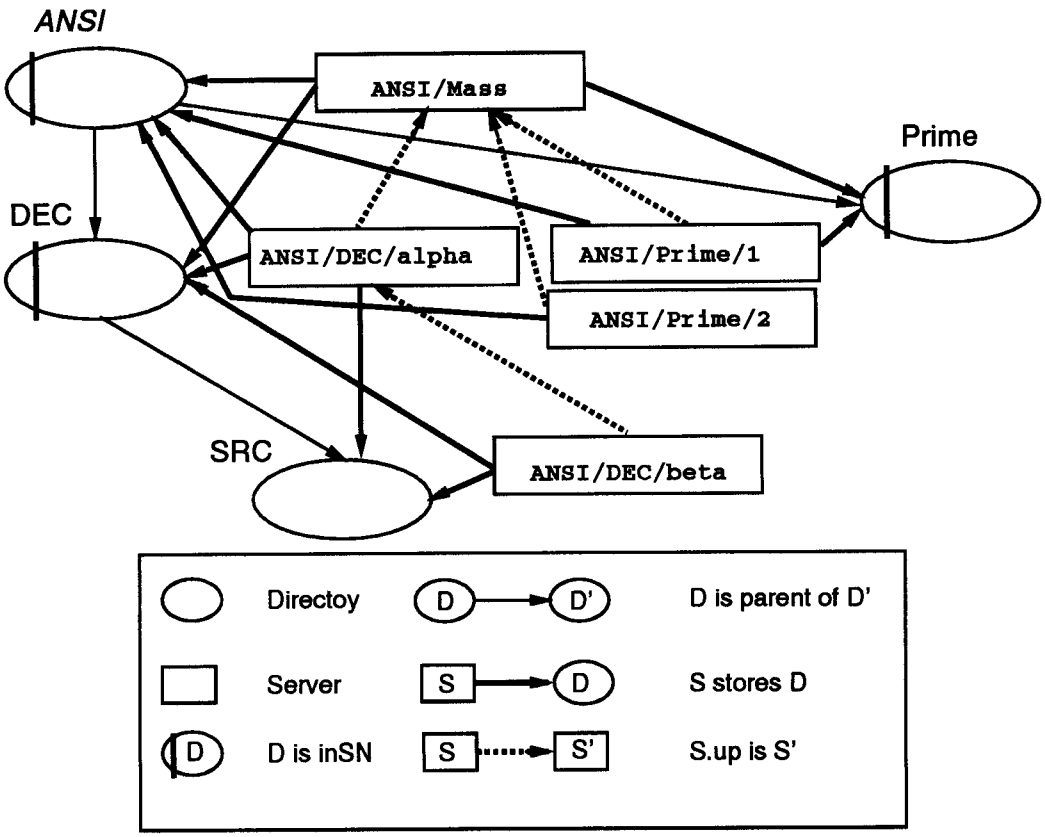


Figure 6.1: Distribution of directories among servers to satisfy the server invariants

that all the replicas form a virtual ring. If failures break the ring, no more sweeps can be done before the ring is repaired. However, changing the virtual ring requires human involvement. Losing some updates which have been done at a single server but not propagated to any of the others is possible. Using heavy replication like this is both expensive for maintenance of consistency and difficult for reconfiguration. Further more, maintaining the ring is independent of maintenance of the replica sets in the GNS. It is up to the administrators to ensure only one ring exists for each replicated directory at a time.

The UNS Approach

In a distributed computing environment, replication can be used to improve performance and availability. By storing copies of shared data on nearby sites, the need for more expensive access to remote sites is decreased. Replication is also the fundamental method for achieving fault tolerance. By replicating critical data on sites with independent failure modes, the probability that at least one copy of the data will be accessible increases. Thus replication approach is used by the UNS. In comparison with the X.500 approach, the UNS provides higher availability and more robust service. In addition, the UNS uses the shorter-server-name approach, i.e. if a server keeps the replica for a directory, the

distinguished name of the server should be shorter than the names of the entries in any directory stored by the server. This is to avoid name resolution loops (see the example in the next section). The UNS does not force a name server to have the replicas of the directories purely for navigation as the GNS, because most of the data replicated in this way is made redundant unnecessarily, and the cost for maintaining consistency is high.

6.3 Dynamic Configuration of the UNS

In Chapter 5, the protocol for the first class name service (UFP) was studied. In this section, the UFP is extended to allow updates to the replica set (defined in Section 5.5.2). The idea of a pseudo-replica is introduced for maintaining consistency of the configuration data. Two invariants are given to ensure safety.

6.3.1 The UNS Configuration

The UNS service configuration has been described in Section 4.2.4. This section follows up Section 4.2.4 with more detail. In the UNS, the naming data is partitioned mainly by authorities which have control over the data. One authority may sponsor several name servers. Some top level naming contexts, such as the root directory of a name space, can be kept by a number of selected authorities within some administrational boundary.

Clients access the UNS via their local User Agents (UAs). On receiving a client request, a UA passes it to a nearby name server. The server may further contact one or more other servers in a recursive way (see Figure 3.3) until it gets either an answer or an error report.

Other kinds of navigations, such as the UA-controlled method (see Section 3.3.3), are not used because a UA does not usually have the right to access every name server involved. It is also difficult to keep the data for communicating with a remote server up to date. The non-recursive server-controlled method is more suitable if underlying multicast mechanisms are provided. However, the UNS considers more general cases.

In Section 4.2.4, three UNS server invariants have been given:

The UNS server invariants :

r1 If $s \in \mathbf{S}$ is not a *root server*, its distinguished name should be shorter than the distinguished names of any entry in any directory it stores.

r2 If $s \in \mathbf{S}$ is not a *root server*, it must store references to other name servers which are

either *root servers* or are *closer* to $\mathcal{G}$.

r3 If $s \in \mathbf{S}$ is a *root server*, it must contain the global index.

As **r2** indicates, the UNS relies on the references for navigations to move up the naming network. There is a trade-off between providing reliable references and hints. Ideally, navigation should be made reliable even when the service configuration is being changed. Caching alone can hardly achieve this requirement. To provide reliable references, however, it must be noticed that although references and replica sets are part of the naming data, they can not be treated simply as ordinary naming data. Thus tools have to be provided at the administration level, which are extension of the tools for maintaining non-configuration data.

An Example

Figure 6.2-(a) contains an example for illustrating how **r1**, **r2** and **r3** are met by the UNS configuration. The rectangle denotes a server, and the oval denotes a directory. The arrow denotes an arc on the directory hierarchy. #XXX represents a unique identifier. The server #101/S1 stores the directory CAMBRIDGE and ENG, the server #/101/ENG/S2 stores SPEECH, #101/S3 stores CAMBRIDGE and CL, and #101/CL/S4 stores SRG and AI. This configuration satisfies invariant **r1**. For instance, the distinguish name of server S2 is #101/ENG/S2, which is shorter than any name with a prefix #101/ENG/SPEECH. If **r1** is broken, a name lookup loop may occur. For example, suppose the server S2 is named #101/ENG/SPEECH/S2. Then it will be impossible to look up this name, since in trying to get from ENG to SPEECH in order to look up S2, the name resolution must resolve #101/ENG/SPEECH/S2; this forms a loop.

Figure 6.2-(b) shows the abstraction of the service configuration, where references are represented by solid lines; other navigations are represented by dashed lines. To meet **r2**, the entries for resolving server name #101/S1 and #101/S3 should be stored by the server S2 and S4 as references. It has been discussed earlier in this chapter that either a piece of knowledge or a full directory may be used for the same purpose. However, the former requires a special treatment and the later uses unnecessary replication. In order to avoid using special method for references, the UFP protocol studied in Chapter 5 is extended to allow a *pseudo-replica* - a reference keeper, to have votes as a full replica. Considering the servers mentioned above again, server S2 and S4 may keep only a reference to S1 and S3 instead of storing a full CAMBRIDGE directory. Therefore the name server group for CAMBRIDGE directory is {S1, S2, S3, S4}, in which both S2 and S4 keep only a pseudo-replica rather than a full-replica.

Both S1 and S3 should also store the global index (see Section 4.2), which is not shown

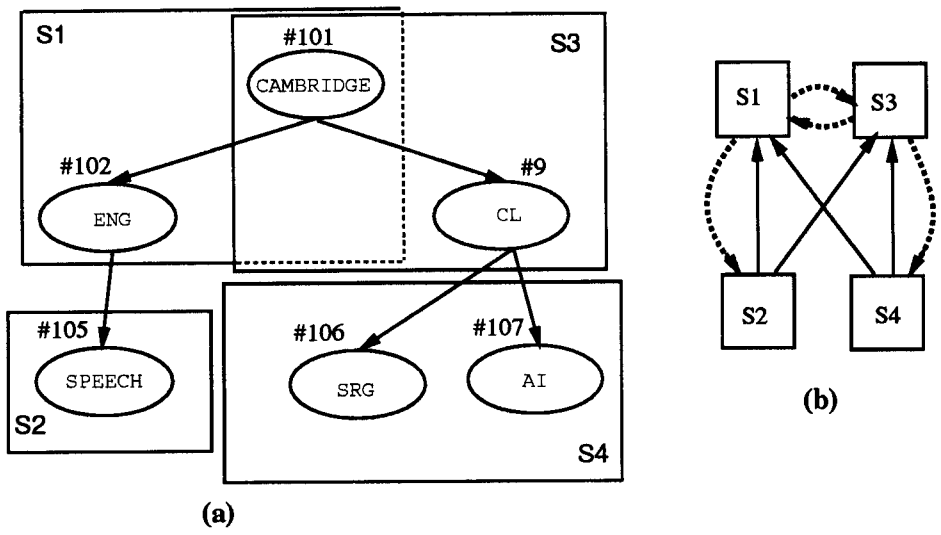


Figure 6.2: A Sample Configuration

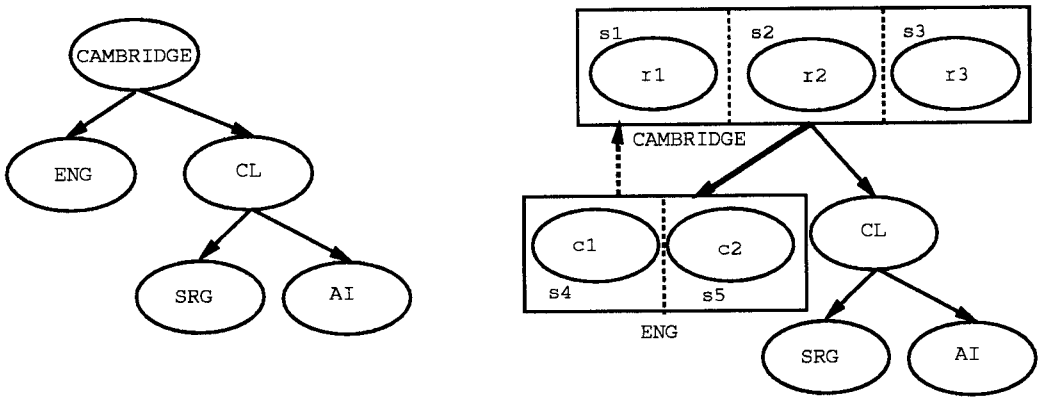
by Figure 6.2.

6.3.2 Maintenance of the UNS Configuration

In previous chapters, the UNS architecture and interface have been defined, and the UNS replication control methods have been described. This section discusses maintenance of the UNS configuration data.

Figure 6.3-(a) illustrates a directory hierarchy and (b) illustrates the same directory hierarchy with replication detail of the CAMBRIDGE and the ENG directory shown in the solid-line rectangles (the replicas are r1, r2 and r3 for the CAMBRIDGE directory; c1 and c2 for the ENG directory). The dashed lines show how the replicas are stored by the servers (the servers are s1, s2, s3, s4, and s5).

As defined in Chapter 3, a directory contains a sequence of entries, each for one of its child directories. An entry contains a number of attributes, one of which is the server name for a child directory. For example, in order to resolve the name CAMBRIDGE/ENG/xyz, the CAMBRIDGE directory is looked up first to get the server name for ENG. The server name is then looked up for an address of the server. After replication is introduced, the server-name attribute should contain a list of server names for all the replicas of a child directory. For example, to look up CAMBRIDGE/ENG, one gets {s4, s5}. Either s4 or s5 can be contacted for the ENG directory. On the other hand, s4 and s5 should both keep a reference (represented by dashed line with an arrow in Figure 6.3-(b)) to the root servers. For example, s4 keeps {s1, s2, s3}.



(a) THE DIRECTORY HIERARCHY

(b) THE DIRECTORY HIERARCHY WITH REPLICATION DETAIL

Figure 6.3: Replication of Directories

In order to run replication protocols, every replica in a name server group (represented in Figure 6.3-(b)) by a solid-line rectangle) should know about the others. This can be done by keeping a list of server names for a replicated directory on each server. For instance, r1, r2 and r3 all keep {s1, s2, s3} to be able to communicate with each other; c1 and c2 both keep {s4, s5} for the same purpose.

Initially the two lists, i.e. the one for navigation stored as an attribute, and the one for replication kept by every replica are identical.

In Chapter 5, the two-class name service infrastructure and the replication control methods which can be integrated to the infrastructure were described in detail. A directory can be implemented based on the infrastructure. Some copies of the directory (called replicas) are stored by the first class servers; some (called read only copies) are stored by the second class servers. The UFP protocol can be run on the first class servers; the epidemic protocols can be run by the second class servers. The question remains of how to store and maintain the configuration data, i.e. the list of server names for a directory's name server group. Before further discussion, a definition is given as follows.

Definition 6.1 Let $\mathbf{N}_d$ denote a server name set which consists of all the names of the servers storing the replicas of the directory d. [1] Let $\mathbf{N}_{d'}$ denote the server name set which consists of all the names of the servers storing the read only copies of the directory d.

To run the UFP protocol, $\mathbf{N}_d$ should be kept by each of the first class servers. Each first class server should also know $\mathbf{N}_{d'}$ in order to call back the second class servers. This can

[1] Logically, when a replica set (see 5.5.2) is changed, the name server group is also changed. Practically, the change of a replica set is carried out by update the server name set, i.e. $\mathbf{N}_d$.

be done by requiring each second class server register with the corresponding first class servers at its creation. There is no need for every second class server to be aware of all the other second class ones, though some of them may cache information about other second class servers. Each second class server, however, should store $\mathbf{N}_d$ to be able to poll the first class servers for refreshing its read only copy and to forward client requests such as updates or "slow read".

Maintenance of the configuration is straightforward because it is similar to that of the ordinary naming data. There is a particular type of entry for representing $\mathbf{N}_{d'}$ on each of the first class servers. The first class servers run the UFP protocol whenever a second class server is to be set up or removed, then inform every member in $\mathbf{N}_{d'}$, including the new member if it is an addition operation, by sending a message containing the update. If any of the second class servers fails to get the message, it may learn it later by polling the first class servers.

Extension to the UFP

The UFP has been discussed in great detail in Chapter 5. It can be easily extended to allow changes being made on $\mathbf{N}_d$. For instance, a new replica of a directory d is installed on the server s, so s should be added to $\mathbf{N}_d$. The change of $\mathbf{N}_d$ is synchronised by the UFP as the client requests. If care is not taken, however, inconsistency may occur. For example, two replica sets for the same directory exist and have no intersection; each appears to be able to form a quorum. In order to avoid this, the extended UFP requires that any request only change one member of $\mathbf{N}_d$.[2] For instance, suppose there are A, B, C, D, E and F in $\mathbf{N}_d$. If for some reason that B and E are to be removed, two requests should be conducted. One may ask what happens if two coordinators start almost simultaneously to remove E and B. The UFP ensures that the two requests will be synchronised one after the other.

It is a little tricky to propagate the change of $\mathbf{N}_d$ to the second class servers. After $\mathbf{N}_d$ on the first class servers is changed, the corresponding copies on the second class servers will become out of date. Again we can rely on the firsts to tell the seconds about the change, but if some of the second class servers miss the information, there could be problems when they try to contact the first class servers later. One solution is to estimate the frequency of changing $\mathbf{N}_d$, and to make each second class server poll the firsts periodically according to the frequency. Since it is not permitted to change more than one member of $\mathbf{N}_d$ on a single request, the second class servers should always know some of the firsts to contact.

Another solution is for some of the second class servers to contact others which probably

[2]Later in this chapter, it is explained further why the restriction is important for the correctness of the UFP.

keep track of $\mathbf{N}_d$. It is very difficult to ensure that every secondary server has an up-to-date $\mathbf{N}_d$ if there are failures. The epidemic algorithms have the property that any update will be propagated eventually so that all replicas are identical if there are no more new updates. Since the change rate of $\mathbf{N}_d$ is very slow, the method is good enough for the UNS.

Configuration Constraints

Correctness constraints, namely three server invariants, have been set in Chapter 4 to ensure that the UNS configuration is coherent initially. These invariants assure that navigation is properly formed, i.e. the name resolution can manoeuvre up and down the naming network. However, during the transition to a new service configuration, it is possible for some service interruption to appear. For instance, a server has been removed, but the references to it still contain the old bindings. Or a new replica has been added, but no one can reach it. In order to prevent the service from interruptions and maintain the robustness of the service, the configuration constraints are introduced.

Definition 6.2 If r is a replica of a directory d, and r is stored on a server s, then $r\mathbf{On}s$ is true.

Definition 6.3 Let $\mathbf{N}_d(P)$ denote the server name set for a directory d, which is kept by d's parent directory as an attribute. Let $\mathbf{N}_d(C)$ denote the server name set stored by every replica of d. Initially, $\mathbf{N}_d(C) = \mathbf{N}_d(P)$.

Definition 6.4 $\mathbf{S}_d = \{s | \exists r \in \mathbf{R}_d \wedge r\mathbf{On}s \text{ is true } \}$,[3] i.e. $\mathbf{S}_d$ is a set of servers keeping replicas for the directory d. There is no distinction between pseudo-replicas and full replicas.

The Two Configuration Invariants

$\mathcal{T}1$: $\mathbf{N}_d(P) \bigcap \mathbf{N}_d(C) \neq \emptyset$.

$\mathcal{T}2$: $\forall s \in \mathbf{S}_d$, s stores $\mathbf{N}_d$ and $\mathbf{N}_{d'}$.

$\mathcal{T}1$ requires that the two server name sets intersect during a configuration change. If $\mathcal{T}1$ is maintained, no interruption of the service navigation will be caused, even if temporary inconsistency between the two sets exists. $\mathcal{T}1$ ensures the correctness of walking down the naming network. If there is only a single replica for directory d, $\mathcal{T}1$ may be broken during a reconfiguration. For example, if the only replica is removed before the parent directory is informed, i.e. $\mathbf{N}_d(C) \bigcap \mathbf{N}_d(P) = \emptyset$. This can cause serious disruption to some part of the system, e.g. a name lookup is sent to a server which does not exist. To

[3]See Section 5.5.2 for replica set.

prevent such disruption from happening, the algorithm should check whether the number of members in a server name set is less than two before a removal operation. However, the problem can not go completely, human intervention may be required if a disruption, which even replication can not help, happens. Alternatively, some underlying support such as multi-site atomic action, or reliable multicast has to be employed.

$\mathcal{T}2$ requires that every server of a name server group keep both $\mathbf{N}_d$ and $\mathbf{N}_{d'}$. Note that although caching can be used to remember some cross references among servers to improve performance, consistency of the two sets should be maintained by the first class servers. $\mathcal{T}2$ ensures that the consistency of $\mathbf{N}_d$ and $\mathbf{N}_{d'}$ are maintained by the first class servers running the UFP protocol. This is important for the correctness of walking up the naming network and for the progress of the UFP protocol.

6.3.3 Discussions

Only two of the operations given in Chapter 3 have effects on the service configuration, i.e. **AddReplica** and **RemoveReplica**. It is worth mentioning that operations on server entries, such as adding a new server, removing a server or moving a server to a new address, do not directly affect the service configuration. For instance, only after all replicas stored by a server have been removed, can the server be removed. The existence of a new server is not so important before it has some data stored on it. Thus name server entries can be treated as ordinary application entries, except they may appear in the non-leaf vertexes of a naming network. [4]

The number of replicas is crucial for choosing replication control protocols. It is obvious that when the number becomes too big, tight replication control methods are expensive. In the UNS, global index should be heavy replicated [5] for availability. The two-class name service infrastructure introduced in Chapter 5 can be used for the index replication control. For example, a handful of sites are nominated as the first class sites, and the rest are the second class sites. The strategy for replicating the index is similar to the UN (United Nation) model. In such a model, permanent members of the security council are like the first class sites; other UN members except those five are like the second class sites. However, unlike the UN, joining the global index or leaving it is allowed if the first class sites agree. It seems politics rather than technology to decide which protocol is used for

[4] It should be noticed that a UNS server has a name shorter than the name of any entries in any directory it stores. Thus the name server object is one of the two types, whose entries may appear in non-leaf vertexes on a naming network. The other is the directory object.

[5] Caching is not considered as replication here.

the index.

For authorisation consideration, a global context (any root other than the imaginary super-root) may not be replicated on servers beyond its authority. A root may be required to be replicated on non-root servers, acting either as reference or for robustness (reliability and availability), or both. For instance, a replica of the UK directory can be kept by a Cambridge server. Directories on higher positions of a naming network are required to be heavily replicated, say kept on some dozens of sites. Again the two-class name service infrastructure can be used. For those directories with fewer expected universal uses, e.g. a university department directory is potentially less interesting than a national directory, fewer replicas are needed, a tight replication control mechanism may be used for offering clients accurate naming data. Loose replication methods may be used too if naming data is very stable. It is important to provide clients flexibility rather than forcing them to commit to only one mechanism.

6.3.4 The Algorithm

In Chapter 5, $\mathbf{R}_d$ was defined as the set of replicas for the directory d, and $\mathbf{S}_d$ the logical Name Server Group (NSG) of d. $\mathbf{N}_d$ is introduced in this chapter for defining the server name set of d. A server may belong to different NSG if it keeps replicas for different directories at the same time. In implementation, $\mathbf{N}_d$ is associated with a timestamp, which is increased after each successful update, and should be kept in a stable storage. Only one replica can be removed, added or moved by a request. This restriction is important for maintaining $\mathcal{T}1$. Because of this, intersection exists between $\mathbf{N}_d(P)$ and $\mathbf{N}_d(C)$ after a reconfiguration.

Chapter 3 has defined the operations for replicas. The interface can be found in Figure 3.6. Here the operations are discussed in more details. The semantics of configuration operations is given informally in this section. The system will be in one of the following status: *service*: when the system can accept and carry out client requests; *suspended*: when the system stops responding ordinary client requests but configuration; *failure*: when the system is not available due to server crashes, communication failures and so on.

The algorithm makes the UNS reconfiguration appear to be atomic to clients by using semantic information and replication. It ensures that an attempted change to the configuration is carried out correctly and becomes stable, otherwise it can be rolled back to the point before the change. Atomicity can be achieved more cheaply than in some well known tight replication control mechanisms. Temporary inconsistency has an effect similar to a

server not responding to client requests due to network delays or other failures. This kind of inconsistency should disappear as soon as the reconfiguration is fulfilled, so that clients only perceive the system running slowly.

GetNset(*dname*, *opt*) simply returns the server name set of the directory *dname*. If "accurate" is specified ($opt = 1$), it is guaranteed that the current state of the server name set is returned. Otherwise the server receiving the **GetNset** request will return the server name set kept by itself.

AddReplica(*sname*, *dname*) adds a replica of the directory *dname* to the server *sname*. This involves installing a replica on the server *sname*, adding *sname* into every copy of the server name set for the directory and modifying the attributes in the directory's parent. When the operation starts, no more client requests are accepted. However, client requests in-progress are allowed to finish. GetNset is used to obtain the server name set for the directory *dname*. The parent directory is updated using the ModifyEntry operation. Note that only after the replica is stored by the server, can its server name set be modified. Before the parent directory is updated, the new replica should be made known to other servers. The parent directory can then be updated. In other words, the operation consists of two synchronisations: the first is for updating the configuration data of replication while the second is for the data of navigation. Thus the server *sname* will remains unknown to the outside world if the operation crashes after the first synchronisation.

RemoveReplica(*sname*, *dname*) removes a replica of the directory *dname* from the server *sname*. Similarly, client requests are not accepted during the operation. The operation updates the corresponding configuration data ($\mathrm{N}_{dname}(P)$) on the parent directory first, then the server *sname* is expelled from $\mathrm{N}_{dname}(C)$. In the end, the replica may be destroyed. In other words, the operation consists of two synchronisations: the first is for updating the configuration data of navigation while the second for the data of replication. Thus client requests will not be passed to server *sname* if the operation fails after the first synchronisation.

Figure 6.4 shows the three procedures. **Nset** is an attribute type for server name sets. The operations do not need multi-site transaction support if the UFP is used to implement the **for** loop in AddReplica or RemoveReplica. ModifyEntry can be implemented with the two-class name service infrastructure.

It is worth mentioning that a directory's server name set is usually different from its parent's. It is important to understand that the two sets are not involved into the same round of synchronisation, i.e. they do not change together.

```
Nset UNS::GetNset(char* dname, int opt)
{
    return(ReadEntry(dname, Nset, opt));
}

int UNS::AddReplica(char* sname, char* dname)
{
    Nset n_d;
    stop accepting client update requests;
    wait until in-progress updates complete;
    install replica of directory dname on server sname;
    n_d ← GetNset(dname,1);
    for (n ∈ (n_d ∪ {sname})) do
    {//change replication configuration
        add dname to server n's server name set;}
    ename ← the name of directory dname's parent;
    //change parent configuration
    ModifyEntry(ename, Nset, n);
    exception handling;
    end
}

int UNS::RemoveReplica(char* sname, char* dname)
{
    Nset n_d;
    stop accepting client update requests;
    wait until in-progress updates complete;
    ename ← the name of directory dname's parent;
    //change navigation configuration
    ModifyEntry(ename, Nset, n);
    n_d ← GetNset(dname,1);
    {//change replication configuration
    for(n ∈ (n_d − {sname})) do
        delete dname from server n's server name set;}
    exception handling;
    end
}
```

Figure 6.4: Configuration Procedures

Correctness of the Algorithm

The algorithm introduced in this chapter allows the replica set to be changed through changing the corresponding server name set. Care must be taken when making such changes, otherwise inconsistency may be caused. For example, there may be a new replica set having a majority that does not intersect with the previous one, so that some client requests already passed get lost. If a majority set of the replica set before the change and after the change intersect with each other, the correctness of the configuration algorithm will be maintained. This can be ensured by requiring that only one member of the server name set be changed by any request.

Definition 6.5: let $\mathbf{M}_b$ be any majority set of $\mathbf{N}_d$ before $\mathbf{N}_d$ is changed. $\mathbf{M}_a$ be any majority set of $\mathbf{N}_d$ after $\mathbf{N}_d$ is changed.

Definition 6.6

Let $\mathbf{N}_b$ be the total number of members in $\mathbf{N}_d$ before operation AddReplica or RemoveReplica, such that

$\mathbf{N}_b = 2k$, if $\mathbf{N}_b$ is an even number,

$\mathbf{N}_b = 2k + 1$, if $\mathbf{N}_b$ is an odd number,

where $k = 1, 2, 3, \ldots$.

Similarly, let $\mathbf{N}_a$ be the total number of members in $\mathbf{N}_d$ after one of the two operations. Suppose that each member has only one vote. The definition implies that $\mathbf{N}_b \geq 2$.

$$\text{Predicate } \mathbf{C}\text{: } \mathbf{M}_b \cap \mathbf{M}_a \neq \emptyset.$$

Predicate **C** means that the majority set of $\mathbf{N}_d$ before a reconfiguration and the one after should intersect. [6]

Lemma: **C** is maintained by the extended UFP.

Proof:

For addition, suppose $\mathbf{N}_b = 2k$, then $\mathbf{M}_b$ has at least $k + 1$ votes, and $\mathbf{M}_a$ has at least $k + 1$ votes because $\mathbf{N}_a = 2k + 1$. If the new member is not in $\mathbf{M}_a$, it is obvious that $\mathbf{M}_b \cap \mathbf{M}_a \neq \emptyset$. If the new member is in $\mathbf{M}_a$, the other k members must have one in common with $\mathbf{M}_b$, otherwise, $\mathbf{N}_b$ should be $2k + 1$ rather than $2k$ □.

Similarly if $\mathbf{N}_b = 2k + 1$, $\mathbf{M}_b$ has at least $k + 1$ votes, and $\mathbf{M}_a$ has at least $k + 2$ votes because $\mathbf{N}_a = 2k + 2$. If the new member is not in $\mathbf{M}_a$, it is obvious that $\mathbf{M}_b \cap \mathbf{M}_a \neq \emptyset$. If the new member is in $\mathbf{M}_a$, then the other $k + 1$ members must have one in common

[6] Majority consensus is used here although the algorithm can also use quorum consensus. For general quorum consensus, dynamic vote reassignment may be involved. Techniques for dynamic vote reassignment have been discussed in [Barbara 86, Barbara 89]. It' s not addressed by this research.

with $\mathbf{M}_b$, otherwise, $\mathbf{N}_b$ should be $2k+2$ rather than $2k+1$ □

For deletion, if $\mathbf{N}_b = 2k$, then $\mathbf{M}_b$ has at least $k+1$ votes, and $\mathbf{M}_a$ has also at least k votes because $\mathbf{N}_a = 2k-1$. The k members forming $\mathbf{M}_a$ must have one in common with $\mathbf{M}_b$, otherwise, $\mathbf{N}_b$ should be $2k+1$ rather than $2k$ □

Similarly, if $\mathbf{N}_b = 2k+1$, then $\mathbf{M}_b$ has at least $k+1$ votes, and $\mathbf{M}_a$ also has at least $k+1$ votes because $\mathbf{N}_a = 2k$. The $k+1$ members forming $\mathbf{M}_a$ must have one in common with $\mathbf{M}_b$, otherwise, $\mathbf{N}_b$ should be $2k+2$ rather than $2k+1$ □

The extended UFP is 1SR because $\mathbf{C}$ is maintained and the UFP is 1SR.

6.4 Summary

The UNS dynamic configuration is discussed in this chapter. Two kinds of naming data are concerned: one for navigation; the other for replication control. Two configuration constraints are introduced to ensure that the UNS configuration is coherent when dynamic configuration is allowed. Operations are developed for managing the configuration, which make use of the extended UFP protocol. A safety condition for changing the replica set is introduced and it is proved that under the safety condition, the extended UFP is 1SR.

As discussed in Chapter 3, although heterogeneity is preferably avoided, it is often not possible to get rid of it completely. Taking international phone systems as examples, observations suggest that uniformity is achieved at national level, while heterogeneity is reserved at the local administration levels. For instance, the phone numbers for two individuals in two different countries may not have the same length, although they both include the three-figure national code. The UNS requires that all participants adhere to some universal agreements such as the format of the distinguished names and the rules of name resolution. However, the UNS also allows for heterogeneous local name services to be interfaced to it. Local naming systems may have different syntax or semantics such as file system naming or RPC naming and binding; they are running with various underlying mechanisms such as multicast or end-to-end network transportation protocols.

Unlike those five configuration methods mentioned in Section 6.2.1, the UNS provides a unified naming scheme. Heterogeneity is allowed at some local levels, but is completely hidden from the client view. It emphasizes scalability, uniformity, availability and robustness.

Chapter 7

Implementation of the UNS

7.1 Overview of The UNS Implementation

A name resolution model, introduced in Chapter 4, allows restructuring of the Universal Name Service (UNS) name spaces. The design of the UNS is described in Chapter 3 and the protocols for naming data maintenance is given in Chapter 5. It is natural to question the feasibility of the sophisticated UNS design. For a system to be practical it must supply a service that clients find useful. The system must be possible to build, and it must perform adequately. This chapter discusses the prototype implementation of the UNS in detail. The rest of this chapter is organised as follows: Section 7.2 discusses how the design given in previous chapters is simplified and what assumptions are made. Section 7.3 describes the UNS prototype implementation in detail. Finally, Section 7.4 discusses what has been learned from the prototype implementation.

7.1.1 UAs and Servers

The UNS can be divided into a number of major modules according to their functions. The UNS consists of two components: the *User Agent* (UA) and the *Server*. A UA runs on each client machine so that a client can invoke a UA operation to pass a request to a name server. A name server maintains a portion of the naming database and processes incoming requests in cooperation with other name servers as necessary. The abstract interfaces for client and server management are described in Chapter 3. A UA has cached information (such as names or addresses) of several name servers, which enables a client to get service from any server available without knowing the server's name or internet address, even when there are some failures of server machines or communication links.

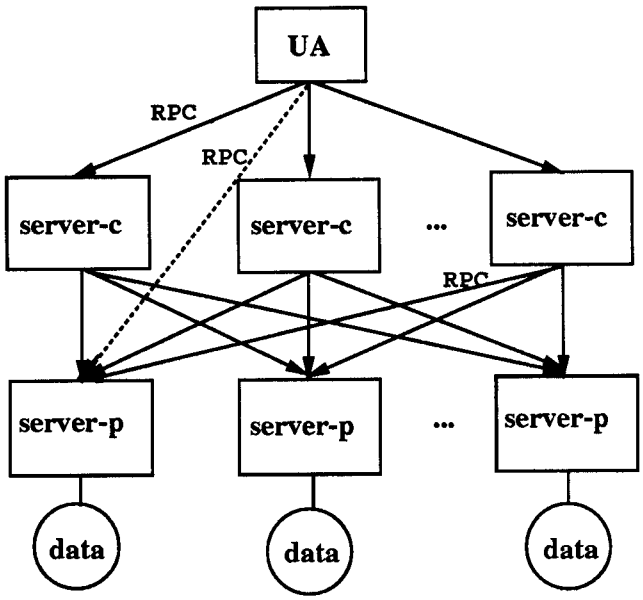


Figure 7.1: The Modules of the First Class Service

Clients invoke UAs by procedure calls, while UAs contact servers through the UA/Server protocol. Servers communicate with each other through the server protocol.

The First Class Service

The two-class name service infrastructure was introduced in Chapter 5. The UNS First Protocol (UFP) and the epidemic algorithms can be integrated into the infrastructure. For running the UFP protocol, defined in Chapter 5, a UNS server should be composed of two parts as shown in Figure 7.1: one performs the coordination function (server-c), and the other performs the participation function (server-p). The participation function has direct access to the naming data. The two parts run in two processes on the same server machine. To implement the *fast read* operation a UA may bypass a server's coordination part and connect to a server's participation part directly. This facilitates good performance, as shown in a dashed line in Figure 7.1.

7.1.2 Clients and Administration

The UNS has a comprehensive set of operations for applications and maintenance. Clients make requests through UAs, while a system administrator can use a special set of operations with direct access to servers for maintaining the service. A distributed monitor service is necessary to give the system administrator an overall view of the current system

state. The provision of such a monitor service is outside the scope of this research.

7.1.3 Storage of Naming Data

The UNS maintains an enormous database partitioned and distributed among name servers. A UNS name can be one of the following types: a UID type, a string type, or a combination of both. A name is mapped onto one or more attributes. Each attribute has a name and a type defined in an object class. Example types for attributes are *title, password, machine address*, and *location*. Operations on each attribute type are defined within the object class to protect object data specified by attribute definitions. A directory object consists of a number of elements and each element is composed of a name to attribute mapping. The size of a directory or an index may not exceed a few megabytes because of name data partitioning. The update rate on each portion of data is low and update operations are not composed of multiple client actions. The simplicity of the UNS database in terms of data type, size, and user demand makes it possible to use the small database technique [Birrell 85]. A server machine might be supported by a stable storage service for information backup.

7.1.4 Configuration

Configuration of the UNS can be done during normal service operation. Dynamic configuration, supported by a group of operations at the service administration level, makes use of the same mechanism as ordinary data. Figure 7.1 does not illustrate how navigation is to be implemented. This may be accomplished by storing proper naming information among several servers, forming an acyclic path towards the target server in which the name to be resolved is bound.

7.2 Prototyping the UNS

Great effort is required to implement a global name service like the UNS. For example, the UK Pilot Project of Directory Service [Kille 89], under development for more than three years, involves many people from a dozen academic institutes. The distinguished features of the UNS are the ability to restructure the name spaces as they grow and the ability to maintain strong data integrity. The prototype UNS focuses on these aspects rather than the implementation of a complete large distributed name service. It is constructed to show the feasibility of the UNS design.

7.2.1 The Simplified Model for Prototyping

Some very large scale name service such as the Domain Name System [Mockapetri88] and the DEC Distributed Name Service [Martin 89] have been running, but they neither support name space restructuring nor the two-class name service infrastructure. The epidemic algorithms has been evaluated at Xerox PARC [Demers 87]. Thus, in order to verify the design proposed by this research, implementation effort for the UNS has been concerned mainly with establishing the first class service maintaining the global index (see Chapter 4). The prototype service should run the UFP developed in Chapter 5 efficiently and support part of the name resolution model described in Chapter 4.

Different Approaches

Two approaches to the implementation of the global index are compared. The main consideration is to make the global index scale gracefully as the Universal Name Service grows. The first approach uses a table of pure names as the unique directory identifiers. Pure names offer the advantage of transparency although, they are not good parameters for location algorithms. The second approach employs a tree structure for a very large global index. Unfortunately, as the system becomes very large the global index must be partitioned, and the system exhibits poorer performance if compared with the former. In a very large system, it is not acceptable to resolve names using frequent global searches. Some compromises must be made to enable the global index to work effectively and efficiently.

In addition to the system scale, a distributed and replicated global index offers greater availability and reliability. A sensitive issue in replicated systems is the propagation of updates. In the UNS design, emphasis is given to different replication control methods in order to meet different consistency requirements of naming data. The index is a common starting point to resolve global names as well as a back up service for clients to find out what is wrong if naming data is stale. The requirements of replicating the global index include: reliability, availability, and fault tolerance.

The most interesting aspect of the implementation is the way in which the index is maintained. As mentioned in Chapter 5, different replication control methods are employed according to different requirements. The index is replicated on many servers. Some servers are the first class servers running the UFP protocol; the others are the second class servers running the epidemic algorithms.

The prototype service provides operations such as *add, disable, modify* and *lookup.* Replication and concurrency control are transparent to clients.

Assumptions Made

In the UNS prototype, a single table implementation is used based on the belief that it can abstract away some common well-known problems of name services, and focus can be given to more interesting problems such as reliability and availability. A table with one thousand entries is implemented. Each entry contains a mapping from a unique identifier to a set of server addresses of a root directory or a set of server identifiers.

Two assumptions are made by the UNS. Firstly, comes the size of the global index. Some tens of thousands organisations around the world may build their own name spaces and be willing to integrate with the UNS. The number of name space roots roughly reflects the number of UDIs in the global index.[1] This requires several hundred thousand to several million bytes of memory, a relatively small amount given current hardware capacity. Therefore, implementing the global index as a table to be stored by a name server is possible. Perhaps, large organisations will have more than one name space. It may also be necessary for some local directories other than the organisational root directories to register themselves with the global index. In other words, it is possible for the size of the global index to grow beyond the specified size. In order to avoid linear growth of the index as the number of UDIs increases, a two level tree structure is proposed to implement the global index. The table approach is very attractive not only because it makes use of pure names to provide a high degree of transparency but also because it can work effectively with the Universal Name Service. With the tree approach, the advantage of using pure names is lost; the UDI name space is partitioned into a root index and several leaf indexes. A UDI must carry some hints of indexes to be resolved efficiently. The advantage of the second approach is that it can accommodate many more UDIs than a single table.

Secondly, comes maintenance of the global index. The index is used to resolve UDIs. Any name lookup of the UNS starts with a UDI. Since the index contains mainly the UDIs of a large number of local root directories, it must be heavily replicated so that the UNS can work efficiently and reliably. Regard to the two methods of implementing the index above, the two-class service infrastructure proposed in Chapter 5 applies to both cases. Some more assumptions are made in the next section to allow the right replication mechanism to be implemented.

[1] UDI stands for the Unique Directory Identifier. See definition in Chapter 3.

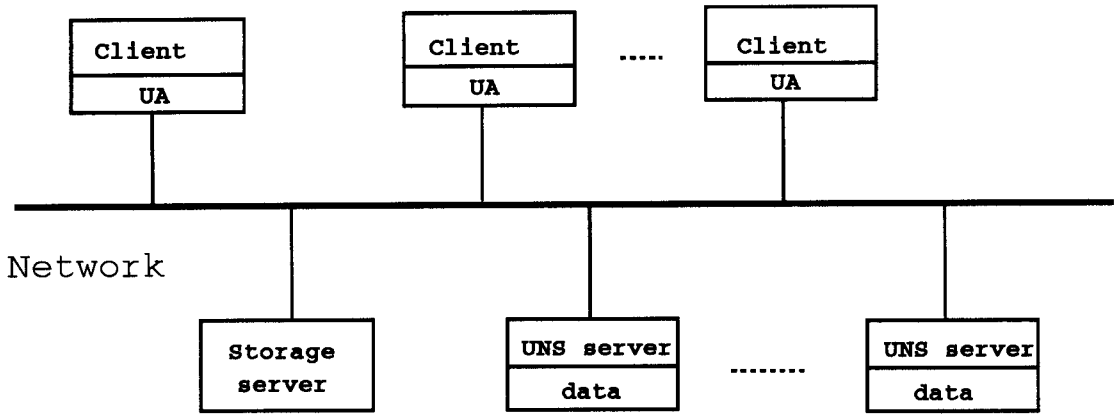


Figure 7.2: The System Environment

7.2.2 The System Environment

Although the name service is designed to be globally accessible, simulations on a LAN are used in this design. Machines involved would be either server or client machine connected by a LAN. A server machine has a considerable amount of main memory and is supported by a stable storage server to retain the server state in the event of a crash. The data is stored in the memory as long as the server remains running. The mechanisms for small databases including logging, check-pointing, and local atomic action are assumed available. Clients may access servers through a User Agent module on any client machine. UAs communicate with servers via RPC and keep caches. Communication among servers is also carried out by RPC. Each machine is equipped with a clock. Figure 7.2 describes the system environment.

Sun RPC

A high-level communication paradigm which enables application programs to make procedure calls in a distributed environment without knowing any details of the underlying network is remote procedure call - RPC mechanism. Many systems providing RPC have been built over the last decade. The Sun RPC [Sun 90] developed in 1985 at Sun Microsystems Inc has three layers. The highest layer is totally transparent to the operating system, machine and network upon which it is run; users are not aware of using RPC at this level. The middle layer hides details about the operating system and other low-level implementation mechanisms; it is distinguished by its simplicity. The lowest layer allows more control over the calls to be made, such as timeout values and specification of transport mechanisms than others. The Sun RPC is available in C programming language.

C++

The C++ programming language [Lippman 89] is an evolution of C with three important extensions:

- support for abstract data use and creation
- support for object-oriented design and programming, and
- support for stronger type checking than C

C++ retains the simplicity of C expression and speed of execution.

Recently, C++ has become available on a variety of platforms. The Cambridge Computer Laboratory makes C++ available on VAX, SUN and HP machines running Ultrix, SunOS, and HP-UX operating systems respectively. Zhixue Wu has written a Sun RPC generator for C++.

7.3 The Implementation

Given the simplified model in section 7.2 and the replication control protocol in section 5.5, this section describes the UNS prototype in detail.

7.3.1 The System Architecture

Following Figure 7.1, the prototype implements a UA class, a UNS class for *server-c* - providing the coordinator function (see also Figure 5.3), and a UFP class for *server-p* - providing the participant function. The index is kept in main memory with a storage server as backup and support for atomic actions. The data structures and their operations provided are described next.

7.3.2 The Data Structure

The data structure of the index is shown in Figure 7.3 using C++ syntax. An index element is composed of a key of the UID type and values associated with the key. The value associated with a key is a VAL type, which is a union of a machine name or a set of SIDs. The index consists of a sequence of elements. As described in Section 5.5, the coordinator issues a synchronisation identifier when starts to synchronise a client request.

The identifier is implemented by the TS class and has three operations: *create, compare* (tcmp) and *increase* (tinc)). The TPR class is for the Table of Pending Requests, which contains *pb* - the identifier of the last synchronisation that the server voted; *nb* - the identifier of the last synchronisation that the server promised to participate; *pd* - the previous request (which are time-stamped) that the server has voted for. The UFP class implements the index and the participant operations. The UNS class implements the replica set, the quorum and the coordinator operations.

7.3.3 The UA Operations

Figure 7.4 outlines the user agent operations. The entrance of the index is well known to any servers. After a Unique Directory Identifier (UDI) is disabled, it may not be used again. There are more operations related to maintenance of the global index such as those to set up an index server, move a server or index etc. Those operations are outlined in Chapter 3.

The semantics of the interface is as follows:

query :

UA::query looks up a given UDI and a name of a server which stores the corresponding directory is returned. [2]

modify :

UA::m_s replaces a specified server name with a new one.

UA::m_d replaces a specified server identifier with a new one.

The modify operations are used to reorganise the name space, e.g. to destroy a directory or a name server, to add a new directory or a server to the UNS, to move a server or a directory, or to merge two existing root directories. The procedure for moving a directory is illustrated in Figure 4.4. The parent of the directory is changed but the UDI for it remains valid. Note, names like #111/C are no longer valid unless a soft link is provided by the system.

To the client, there appears to be a single index, although the implemented index is replicated. Any operation can raise a "failure" exception, which means the service is unavailable even with replication. A full description of the algorithms used for replication control was given in Chapter 5.

[2]The RPC mechanism translates a machine name to a network address and uses the result to make a connection to a remote machine.

```
class UID {
    public:
        UID create(void); int uidcmp(UID);
    private:
        long UID; };
class VAL {
    int type;
    union {
        char ad[CSIZE];
        long sid[NUM_REP];
    } VAL_u; };
class Element {
    UID udi; VAL val; };
class TS {
    public:
        TS create(void); int tcmp(TS); TS tinc(TS);
    private:
        long t; UID site; };
class D {
    short flag; TS dn;
    char d[LENGTH];
    UID v0; VAL v1; VAL v2; }
class LV {
    TS b; int vote; D d; };
class TPR {
    TS pb; TS nb; D pd; };
class UFP {
    public:
        void Init(UID, int); LV NextS(TS,TS);
        int BeginS(TS,D); VAL Commit(TS,D);
    private:
        TS ver; TPR tpr[TSIZE]; RW lock;
        FILE index_log;
        Element index[SIZE];
        VAL _lookup(UID); int _modify(UID, VAL,VAL);
        int _insert(UID); int _disable(UID); };
```

Figure 7.3: Object Class of the Index and Others

UA Operations

```
int UA::query(UID key, int opt, char* a) {
    exceptions: NotFound,
                Failures;
};
int UA::m_s(UID key, char* vn) {
    exceptions: NotFound,
                NotCompatable,
                ConfigurationError,
                Failures;
};
int UA::m_d(UID key, UID vo, UID vn) {
    exceptions: NotFound,
                NotCompatable,
                ConfigurationError,
                Failures;
};
//For administration only
LIST UA::Snapshot(void) {
    exceptions: Failures;
};
int UA::Sconf(char *old, char *new) {
    exceptions: NotFound,
                Failures;
};
```

Figure 7.4: UA Operations

7.3.4 The Server Operations

The operations provided by servers include *snapshot* for replica set, a set operations for the UFP, as well as *lookup* and *modify* operations. The server also provides operations for failure recovery, which are not shown in figure 7.5. The operations are at the system level and are transparent to clients.

The semantics of the server operations is as the following, see also Section 5.5 for the detail of the UFP:

UFP::NextS :

This operation checks whether the coordinator has the current data. During the first phase of the UFP, the coordinator (implemented by UNS::svc_c) invokes **UFP::NextS** several times to send its version number and a synchronisation identifier to every active server belongs to the same NSG - the Name Server Group. An "OK" is returned if the version is current, otherwise the participant's current version number is returned. A "REJ" is returned if the coordinator does not start with a big enough synchronisation identifier.

UFP::BeginS :

This operation is for synchronising the client requests. During the first UFP phase, on receiving a quorum number of votes from some participants, the coordinator can choose a client request. Then it begins the second UFP phase by calling **UFP::BeginS** several times to send the chosen request to every server which voted for the **UFP::NextS**. Either a vote (represented by an integer) is returned by the participating server or an error is raised.

UFP::Commit :

This operation tries to implement the client requests. On receiving a quorum number of votes from some participants, the coordinator invokes **UFP::Commit** with the chosen request and the incremented version number. If it is a lookup operation, a value will be returned. If it is an update operation and the update is successful, zero is returned. Otherwise an error is raised.

UFP::_lookup :

This operation implements the low level operation for lookup. It looks up a given UDI. If the UDI exists, a server name for the corresponding directory or a SID for a server storing the directory is returned. Otherwise errors are raised.

UFP::_modify :

This operation implements the low level operation for modification. It replaces a specified server name or identifier with a new one.

UNS::svc_c :

This operation runs the coordinator code, which invokes UFP::NextS, UFP::BeginS and UFP::Commit respectively during the three UFP phases.

UNS::Snapshot :

This operation returns the current member of a NSG. Errors are raised if servers are unavailable or there are communication failures.

UNS::Sconf :

This operation allows dynamic change of a NSG. Only one server can be added or removed at a time. Errors are raised if the operation is not successful.

7.4 Lessons from the Implementation

The feasibility of the UNS design has been verified in part by prototyping the global index. This section presents the analysis of the performance results which is followed by a discussion.

7.4.1 Performance Measurement

Performance measurements of the UNS prototype have been taken on a number of DEC microVAXII and DEC vs2000 workstations under light loading (between 0.5-1.4). The round-trip delay of Sun RPC using C++ is about 18 ms. Lookup on the local Ethernet with preference to fast read is satisfied in 25-54 ms. The elapsed time for running UFP is about 100-130 ms. Table 7.1 gives the elapsed time for lookup and modify operations with a varying number of replicas. The increase in cost as the number of replicas increases is due to sequential RPC calls in the current implementation. If a multi-threading or other mechanism were used, the linear increase could disappear. The single site performance measurement in Table 7.1 is in fact meaningless but is illustrated to estimate the performance improvement should a parallel RPC call be employed.

7.4.2 Response to the Design Objectives

In Section 3.1.2, the UNS design objectives were outlined. The advantages of the UNS are its ability to restructure name spaces and support of high availability along with strong

Server Operations

```
LV UFP::NextS(TS b, TS num) {
    exceptions: Failures;
};
int UFP::BeginS(TS b, D dec) {
    exceptions: Failures;
};
VAL UFP::Commit(TS b, D dec) {
    exceptions: Failures;
};
VAL UFP::_lookup(UID key) {
    exceptions: Failures;
};
int UFP::_modify(UID key, VAL old, VAL new) {
    exceptions: Failures;
};
int UNS::svc_c(UID key, VAL old, VAL new) {
        Failures;
}; LIST UNS::Snapshot(void) {
    exceptions: Failures;
};
int UNS::Sconf(VAL old, VAL new) {
    exceptions: NotFound,
        Failures;
};
```

Figure 7.5: Some Server Operations

The number of replicas	operations	
	lookup (ms)	modify (ms)
1	125	127
3	246	314
7	510	721
10	727	998

Table 7.1: Performance Measurement

naming data integrity. The UFP protocol is developed for replication control of the UNS first class name servers.

- Flexibility
  The UNS design supports flexibility of restructuring name spaces naturally through merging or migration. The prototype implements the global index on a LAN without any practical difficulty. Merging name spaces and migrating directories or servers are both supported.

- High availability and strong data integrity
  Usually a lookup request from a client can be satisfied by a *fast read* (*local*) operation in less than 60 ms (elapsed time 25-54 ms). A local lookup operation is much faster than a *quorum read* (elapsed time: 246-727 ms). Clients can specify preference to a query in order to meet their needs. The prototype shows that the objective of providing strong naming data integrity can be achieved at a reasonable cost.

- Dynamic configuration
  The prototype supplies operations for UA or name server set reconfiguration. Reconfiguration is done by the system administrator via the administration interface.

- Fault tolerance
  The modularity implemented by the UNS prototype shows that failures are isolated. If a client fails, servers are not affected. If a server fails, other servers will continue. A server-c can not run without a server-p, so that if a server-p is down, the corresponding server-c fails as well. Clients are completely unaware of the connection with a particular server, chosen by the UA at the time the connection is established; if a server or communication link fails, a UA will find another available server and try to contact it. As far as the UFP is concerned, if the required quorum number of servers are alive, the service continues. Hence, the failure of a few sites will certainly not cause the entire system to halt.

It is highly valuable to allow multiple coordinators. Experiments with the UNS prototype show that a higher degree of concurrency is obtained when there are two coordinators organising the synchronisation simultaneously; each runs as if it were the only coordinator. Performance decreased when more than two coordinators issued requests simultaneously. The implementation ensures that no fault is caused in this case but a server might be suspended and a client gets a " server not available, try later" message. The decision to allow multiple coordinators is feasible and avoids the complexity of having an election algorithm.

The scale of the implementation is small in comparison with the scale implied by the real UNS. Some properties of the UNS such as those mentioned in Chapter 5 can only be validated by a large scale implementation.

The current implementation mechanism can be easily applied to realize a global index server containing up to several millions elements. There is no clear indication, however, that the UNS will grow too big so that the tree structure is required to implement the global index. In the future, when mobile computing comes to daily life, and organisations are no longer the major base of the directories, the tree structure or other mechanisms may be required.

Chapter 8

Conclusions

This research has investigated several issues of designing a universal name service. The UNS provides primitive naming facilities to support sharing of objects and communication among various entities in a distributed computing environment. In addition to naming objects unambiguously and accessing objects by name efficiently, the UNS is very flexible to adjust itself to reflect organisational changes. Although existing database management techniques as well as communication protocols, including RPC mechanisms, can be employed by name services, the thesis points out that the semantics of naming should also be exploited for better performance and service reliability.

This chapter concludes the research presented by illustrating three parts: firstly, it reviews what has been done in the area, and motivates the UNS design; secondly, it compares the UNS with other approaches; successes and losses are analysed; finally, it reveals the possibility to set up an Open Name System Architecture, which allows applications of name services freedom to combine or to dispose of naming components with various functions.

8.1 Towards Universal Naming

The advantages of distributed systems have been widely recognised. A distributed system can be extended incrementally to meet the growing demands from its customers. Sharing in such systems has reached the greatest degree ever since computers were invented. A distributed system is of higher availability and more fault-tolerant. Reliability can also be achieved even when there are failures.

Although a distributed system may be built in a more cost-effective way than a conventional centralised system, they have some disadvantages. Leslie Lamport once defined a

distributed system as a system in which the failure of a computer one has never heard of can make it impossible to get work done. A centralised system provides a high degree of coherence since all resources can be used and managed in a uniform way. The real challenge today is not only how to exploit the benefits of a distributed system but also how to maintain its coherence. The designers of distributed naming systems face this and other challenges.

As reviewed in Chapter 2 and Chapter 3, previous work has contributed significantly to computer naming in the following aspects:
[Shoch 78, Saltzer 79, Saltzer 82, Birrell 82, Oppen 83, Sollins 85, Terry 85, Lampson 86, Schwartz 87, Mockapetri88, Peterson 88, Cheriton 89, Martin 89, Linden 90]

- purposes of naming: to support cooperation, communication and sharing
- name resolution models and mechanisms
- unambiguous naming
- name database maintenance
- five naming models: uniform naming, federated naming, naming confederation, integrated naming and heterogeneous naming
- distinct functions of naming: attributes-based naming, primitive naming, addressing and routing
- exploiting replication and caching

However, there are still many unanswered questions. For instance, scale and evolution of distributed naming systems, configuration management, and reliability with high availability. Dissatisfaction with existing naming systems motivated the design of the UNS.

8.2 Evaluation of the UNS

Based on the experience from the UNS prototype, the design goals given in Section 3.1.2 have been met. The main contributions of the research include:

1. The construction of a *global UDI name space*. Flexibility of name space restructuring is supported by allowing the addition of directories to or the removal of directories from the global UDI name space. Hierarchical name resolution is also supported by

the UNS because using unique identifiers only can not satisfy other requirements such as autonomy;

2. a two-class name service infrastructure is defined for replication control. Data replication implies storage of shared data on processors with independent failure modes; it improves performance and availability because the need for expensive, remote read accesses is decreased, and the probability of at least one copy of the data will be available increases [Davidson 89]. In the UNS, the distinction of replicated data is made. Some are defined as *replicas* which are managed by a few number of servers called first class servers. Others are defined as *read-only copies* maintained by secondary servers. The rest are *client-caches*. Some secondary servers for a data partition could be the first class servers for other partitions of naming data. This distinction makes the following possible: replicas are responsible for UNS data integrity; read-only copies are responsible for higher availability; caching is used to improve performance. The second class servers reduce read overhead from the first class servers, and improve service efficiency when most application requirements can be satisfied by making use of hints;

3. this research indicates that name service configuration involves two distinct issues: how to store and maintain replication configuration; and how to store and update server configuration. Previous work has addressed one or the other of these concerns but not both. The UNS takes both into account and novel methods are used to support dynamic service configuration.

Standardisation vs. Heterogeneity: Although based on the uniform naming approach, the UNS design differs from both uniform naming and heterogeneous naming, each of which falls into one of the extreme end of a design spectrum. In fact, arguments on standardisation and heterogeneity may continue, but it seems that neither is satisfactory by itself. For example, even the HNS [Schwartz 87] has to impose a degree of standardisation - the scheme only works if the context space is administered in a centralised fashion. On the other hand, heterogeneity is not allowed by most global name services, such as the GNS and the DNS [Lampson 86, Mockapetri88]. It is valuable for a large name service to let its clients access the service from anywhere in a coherent way, which also makes management simple. In fact, compromises between the two approaches can be made when a global name service is designed. For example, particulars which make a local name service attractive can be reserved by leaving the resolution of relative names for individuals to local servers. The UNS imposes a global naming scheme for the DNs, while allowing certain extent of heterogeneous naming to INs. The UNS approach can be justified by the fact that it

reflects the objectives of distributed systems and reserves system coherence.

Using Globally Unique Identifiers: Some naming system designers concluded that globally unique identifiers are neither useful nor practical; the naming scheme does not scale well [Sollins 85, Stroud 88, Cheriton 88]. Stroud argues that if two independent distributed systems using globally unique identifiers want to merge, it is difficult to ensure that identifiers do not clash. For example, suppose that two LOCUS systems [Walker 83] are combined into a larger one. There is no way to avoid name conflicts. The design and implementation of the UNS show that global unique identifiers (i.e. the UDIs or USIs or GUDIs defined in Chapter 3 and Chapter 4) are very useful in restructuring name spaces. Furthermore, the way to manage them is straightforward. The scaling problem is solved by creating a Global UDI name space, along with a GUDI resolution mechanism. It is not necessary for every individual to have a GUDI. Only root directories, or directories on the higher levels of a naming hierarchy are managed through the GUDI name space. The GUDI name space has a low rate of change, and is expected to maintain tens of millions of name spaces or high level naming contexts as the system grows continuously. Although in principle, the scaling problem is still there, in practice, no scenario such that a system outgrows the universal context is seen.

The GNS [Lampson 86] can be extended or restructured similar to the UNS. However, it does not solve the scalability problem as mentioned in Chapter 4. On the other hand, it is very difficult for the GNS to provide precise naming information like the UNS. The GNS supports authentication without global trust while the UNS does not. However, there should not be any significant difficulty for the UNS to do so.

Federated Naming vs Global Naming: Federated architectures offer freedom of association and a high degree of autonomy [Heimbigner85, Linden 90]. The UNS has a different goal compared with a federated naming system. Firstly, it emphasizes global coherence in the support of applications such as electronic mail, global file systems, and authentication systems without global trust. Secondly, it provides a naming system with the ability to grow incrementally, rather than the modest size assumed with federated naming. Thirdly, it improves service efficiency by employing a less general naming scheme than a federated naming system.

Replication Control Methods: In Chapter 5, how to integrate the UFP protocol and the epidemic algorithms into the two-class name service infrastructure were discussed. On the one hand, the UFP can provide reliable data at a reasonable cost and no multi-site atomic action support is required. On the other hand, the epidemic algorithms are simple and require a few guarantee from the underlying communication system. Thus the UNS

allows a query to proceed at any time, while maintaining reliable naming data. However, the two-class service infrastructure and the replication control protocols should not be forced to every portion of naming data. For instance, one class name service may be run at some site where replication is not necessary. A master/slave protocol may well be run on a local name service with a reliable multicast support. The master/slave strategy can also be seen as an extreme example of the two-class name service infrastructure, where only one server is the first class server and the others are the second class servers. The naming semantics of various systems may be best reflected if multi-class name servers running multiple replication control protocols can be set up.

Experiences with the UNS prototype have suggested that the decision to have multiple coordinators is feasible and avoids the complexity of having an election algorithm. A shortcoming of having multiple coordinators is progresses may be slow. However, no scenario has suggested that the frequency of operations on the first class servers are high enough to prevent progresses of the service from happening.

8.3 Suggestions for Further Research

As computer networking and hardware technology develop rapidly, the potential exists for any computer to communicate with millions of others - anywhere in the world. The Open System Interconnection (OSI) [Zimmerman 80] proposes a suite of protocols for data transport, authentication of communicating entities, file transfer, and remote terminal accesses. In order to meet the need of the OSI, further investigations into computer naming should include the construction of an *Open Naming System Architecture (ONSA)*, which consists of the following components: descriptive naming, secure naming, primitive naming, federated naming and lower level naming.

The ONSA should be generic and instantiated to provide a particular name service. The ONSA allows great flexibility and efficiency because of its modularity. Designers may choose what to have when building a naming system. Clients may also invoke interfaces at various levels accordingly.

It is not easy to generalise computer naming for different purposes, e.g. naming of file systems, naming and binding of RPCs or mail addressing. Contradictory conclusions were drawn from several different approaches. For instance, globally unique identifiers are proven useful for name space restructuring in the UNS, but are not useful or implementable for others. Criteria need to be established so that useful tradeoffs can be made between the existing choices.

Although a small scale prototype has been produced here, the design criteria relate to a large scale system. Only large-scale practical implementation can fully validate the ideas.

Appendix A

The Paxon Synod Protocol

The Synod's decree was chosen through a series of the numbered *ballots*, where a ballot was a referendum on a single decree. In each ballot, a priest had the choice only of voting for the decree or not voting. Associated with a ballot was a set of priest called a *quorum*. A ballot succeeded if and only if every priest in the quorum voted for the decree.

Paxon mathematics defined three conditions on a set $\mathcal{B}$ of ballots, and then showed that consistency was guaranteed and progress was possible if the set of ballots that had taken place satisfied those conditions. The first two conditions can be stated informally as follows.

B1($\mathcal{B}$) Each ballot in $\mathcal{B}$ has a unique ballot number.

B2($\mathcal{B}$) The quorum of any two ballots in $\mathcal{B}$ have at least one priest in common.

The third condition was more complicated. One Paxon manuscript contained the following, rather confusing, statement of it.

B3($\mathcal{B}$) For every ballot B in $\mathcal{B}$, if any priest in B's quorum voted in an earlier ballot, then the decree of B equals the decree of the latest of those earlier ballots.

The Basic Protocol

Steps 1-6 describe how a single ballot is conducted by a priest p, who is called the president. Each priest p had to maintain the following information in the back of his ledger:

$lastTried[p]$ The number of the last ballot that p tried to initiate, or $-\infty$ if there

was none.

$prevVote[p]$ The vote cast by p in the highest-numbered ballot in which he voted, or $-\infty$ if he never voted.

$nextBal[p]$ The largest value of b for which p has sent a $LastVote(b, v)$ message, or $-\infty$ if he has never sent such a message.

1. Priest p chooses a new ballot number b greater than $lastTried[p]$, sets $lastTried[p]$ to b, and sends a $NextBallot(b)$ message to some set of priest.

2. On receiving a $NextBallot(b)$ message from p with $b > nextBal[q]$, priest q sets $nextBal[q]$ to b, and sends $LastVote(b, v)$ message to p, where v equals to $prevVote[q]$ (A $NextBallot(b)$ message is ignored if $b \leq nextBal[q]$.)

3. After receiving a $LastVote(b, v)$ message from every priest in some majority set Q, where $b = lastTried[p]$, priest p initiates a new ballot with number b, quorum Q, and decree d, where d is chosen to satisfy **B3**. He then sends a $BeginBallot(b, d)$ message to every priest in Q.

4. Upon receipt of a $BeginBallot(b, d)$ message with $b = nextBal[q]$, priest q casts his vote in a ballot number b, sets $prevVote[q]$ to this vote, and sends a $Voted(b, q)$ message to p. (A $BeginBallot(b, d)$ is ignored if $b \neq nextBal[q]$)

5. If p has received a $Voted(b, q)$ message from every priest q in Q (the quorum for ballot number b), where $b = lastTrid[p]$, then he writes d (the decree of that ballot) in his ledger and sends a $Success(d)$ message to every priest.

6. Upon receiving a $Success(d)$ message, a priest enters decree d in his ledger.

Bibliography

[Alsberg 76] P. A. Alsberg and J. D. Day, "A principle for Resilient Sharing of Distributed Resources", In *Proc. of the 2nd Intl. Conf. on Software Engineering*, pages 627–644, 1976.

[Araujo 88] R.B. Araujo, "The Architecture of the Prototype COSMOS Messaging System", In *EUTECO'88*, pages 157–169, 1988.

[Bacon 87] J. M. Bacon and K. G. Hamilton, "Distributed Computing with RPC:the Cambridge Approach", Technical report, Computer Laboratory, Cambridge University, TR 117, 1987.

[Barbara 86] D. Barbara et al., "Protocols For Dynamic Vote Reassignment", In *The 5th ACM Symp. on Principles of Distributed Computing*, 1986.

[Barbara 89] D. Barbara et al., "Increasing Availability Under Mutual Exclusion Constraints with Dynamic Vote Reassignment", *ACM Transactions on Computer Systems*, 7(4):394–426, 1989.

[Benford 88] S. D. Benford, *Research into the Design of Distributed Directory Services*, PhD thesis, Dept of Computer Science, University of Nottingham, 1988.

[Bernstein 87] P.A. Bernstein et al., *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[Birrell 82] A.D. Birrell et al., "Grapevine: An Exercise in Distributed Computing", *Comm. of the ACM*, 25(4):260–273, April 1982.

[Birrell 85] A. D. Birrell et al., "A Simple and Efficient Implementation for Small Databases", Technical report, DEC System Research Centre, RR 24, Jan 1985.

[Birrell 91] A. D. Birrell et al., "The Echo Distributed File System", Technical report, DEC System Research Centre, Position paper, mar 1991.

[Bloch 82] J. Bloch et al., "A Weighted Voting Algorithm for Replicated Directory", *Journal of the ACM*, 34(4):859–909, October 1982.

[Cheriton 88] D.R. Cheriton, "The Role of Domains in Large-Scale Distributed Systems", In *the European SIGOPS Workshop*, 1988.

[Cheriton 89] D. R. Cheriton and T. P. Mann, "Decentralising a Global Naming Service for Improved Performance and Fault Tolerance", *ACM Trans. on Computer Systems*, 7(2):147–183, 1989.

[Comer 87] D. E. Comer and L. L. Peterson, "Names and Name Resolution", In B. K. Bhargava, editor, *Concurrency Control and Reliability in Distributed systems*, pages 489–524. Van Nostrand Reinhold Company, New York, 1987.

[Comer 88] D. Comer, *Internetworking with TCP/IP: Principles, Protocols and Architecture*, Prentice-Hall Intl, 1988.

[Comer 89] D. E. Comer and L. L. Peterson, "Understanding Naming In Distributed Systems", *Distributed Computing*, 3(2):51–60, 1989.

[Coulouris 88] G. F. Coulouris and J. Dollimore, *Distributed Systems, Concepts and Principles*, Addison-Wesley, 1988.

[Davidson 89] S. B. Davidson, "Replication Data and Partition Failures", In *Distributed Systems (S. Mullender Ed.)*, pages 265–292. Addison-Wesley Publishing Company, 1989.

[Demers 87] A. Demers et al., "Epidemic algorithms for replicated database maintenance", In *Proc. of the 6th ACM Symp. on Prin. of Distributed Computing*, pages 1–11, 1987.

[El-Abbadi 85] A. El-Abbadi et al., "An Efficient Fault-tolerant Protocol for Replicated data Management", In *Proc. of the 4th ACM Symp. on Principles of Database Sys.*, pages 215–229, 1985.

[El-Abbadi 86] A. El-Abbadi and S. Toueg, "Maintaining Availability in Partitioned Replicated Databases", In *Proc. of the 5th ACM Symp. on Principles of Database Sys.*, pages 240–251, 1986.

[Estrin 86] D. Estrin, *Access to Inter-organisation Computer Networks*, PhD thesis, LCS, Massachusetts Institute of Technology, 1986.

[Gifford 79] D.K. Gifford, "Weighted Voting for Replicated Data", In *Proc. of the 7th ACM Symp. on Operating System Principles*, pages 150–162, 1979.

[Heimbigner85] D. Heimbigner and D. Mcleod, "A Federated Architecture for Information Management" *ACM Trans. on Office Information Systems*. Lab of Computer Science, Cambridge, MA, 1988.

[ISO86] "Open Systems Interconnection: Specification of Abstract Syntax Notation One(ASN.1)", Technical report, ISO/DIS 8824.2, International Standard Organisation, May 1986.

[Kille 89] S.E. Kille, "The Design of QUIPU (version 2)", Technical report, University College London, RN/89/19, 1989.

[Ladin 88] R. Ladin et al., "A Technique for Constructing Highly-Available Services", *Algorithmica*, (3):393–420, 1988.

[Ladin 90] R. Ladin and B. Liskov, "Lazy Replication: Exploiting the Semantics of Distributed Services", In *Proc. of the 9th ACM Sym. on Prin. of Distributed Computing*, Quebec, Canada, Aug 1990.

[Lamport 82] L. Lamport et al, "The Byzantine General Problem", *ACM Transactions on Programming and Language Systems*, 4(3):383–401, 1982.

[Lamport 89] L. Lamport, "The Part-Time Parliament", Technical report, DEC System Research Centre, TR 49, 1989.

[Lampson 81] B. W. Lampson, "Atomic Transactions", In *Distributed Systems* - Architecture and Implementation *(B. W. Lampson et al. Ed.), LNCS 105*, pages 246–265. Springer Verlag, 1981.

[Lampson 86] B.W. Lampson, "Designing a Global Name Service", In *Proceedings of the 5th ACM Symposium on Prin. of Distributed Computing*, 1986.

[Linden 90] R. J. van der Linden, "Federated Naming Model", Technical report, Architecture Project Management Ltd., RC.216.01, 1990.

[Lippman 89] S. B. Lippman, *C++ Primer*, Addison-Wesley Publishing Company, 1989.

[Liskov 88] B. Liskov, "Distributed Programming in Argus", *Comm. of the ACM*, 31(3):300–312, March 1988.

[Mann 87] T.P. Mann, *Decentralised Naming in Distributed Computer Systems*, PhD thesis, Dept of Computer Science, Stanford University, 1987.

[Mann 89] T. Mann et al., "An Algorithm for Data Replication", Technical report, DEC System Research Centre,RR46, Jun 1989.

[Martin 89] S.J.Martin et al., "Development of the VAX Distributed Name Service", *Digital Technical Journal*, (9):9–15, June 1989.

[Mockapetri88] P.V. Mockapetris and K.J. Dunlap, "Development of Domain Name System", In *ACM SIGCOMM'88 Symp. of Comm. architectures and protocols*, California, Aug 1988.

[Mullender 87] S.J. Mullender (Ed.), *The Amoeba Distributed Operating System: selected papers 1984-1987*, CWI, 1987.

[Needham 82] R. M. Needham and A. J. Herbert, *The Cambridge Distributed System*, Addison-Wesley, 1982.

[Needham 88] R.M. Needham, "Naming and Protection", In *Preliminary lecture notes of the Arctic'88, participants edition*, Tromso, Norway, July 1988.

[Oppen 83] D.C. Oppen and Y.K. Dalal, "The Clearinghouse: A Decentralised Agent for Locating Named Objects in a Distributed Environment", *ACM Transactions on Office Information Systems*, 1(3):230–235, July 1983.

[Pandhi 87] S. N. Pandhi, "The Universal Data Connection", In *IEEE Spectrum*, pages 31–37, July 1987.

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

[Quarterman86] J.S. Quarterman and J.C. Hoskins, "Notable Computer Networks", *Communications of the ACM*, 29(10):993–971, October 1986.

[Saltzer 79] J.H. Saltzer, "On the Naming and Binding of Objects", In *Operating Systems - An Advanced Course (R. Bayer et al. Ed.)*, pages 99–208. Springer Verlag, 1979.

[Saltzer 82] J.H. Saltzer, "On the naming and binding of network destinations", In *Proceedings IFIP/TC6 International Symposium on Local Computer Networks*, pages 311–317, Florence, Italy, 1982.

[Schlichtin83] R. D. Schlichting and F. B. Schneider, "Fail-Stop Processors: An approach to Designing Fault-Tolerant Computing Systems", *ACM Trans. on Computer Systems*, 1(3):222–238, August 1983.

[Schroeder 84] M.D. Schroeder et al., "Experience with Grapevine", *ACM Transactions on Computer systems*, 2(1):3–23, February 1984.

[Schwartz 87] M.F.Schwartz, *Naming In Large, Heterogeneous Systems*, PhD thesis, University of Washington, 1987,
Available as technical report 87-08-01.

[Shoch 78] J.F. Shoch, "Inter-network naming, addressing and routing", In *Proc. of IEEE COMPCON Fall 78*, pages 72–79, sept 1978.

[Sollins 85] K. R. Sollins, *Distributed Name Management*, PhD thesis, LCS, Massachusetts Institute of Technology, 1985,
Available as technical report MIT/LCS/TR-331.

[Solomon 82] M. Solomon et al., "The CSNET Name Server", *Computer Networks*, 6(3):161–172, 1982.

[Stefano 87] C. Stefano and G. Pelagatti, *DISTRIBUTED DATABASE - Principles and Systems*, McGraw-Hill Book Company, 1987.

[Stonebrake79] M. Stonebraker, "Concurrency Control and Consistency of Multiple Copies of data in Distributed INGRES", *IEEE Transactions on Software Engineering*, 5(3):188–194, 1979.

[Stroud 88] R. Stroud, "Autonomy or Interdependence in Distributed Systems?", In *the European SIGOPS Workshop*, 1988.

[Sun 90] Sun Microsystems, "Network Programming", In *Network Programming Guide*, 1990.

[Terry 85] D.B. Terry, *Distributed Name Servers: Naming and Caching in Large Distributed Computing Environments*, PhD thesis, University of California, Berkeley, 1985,
Available as Xerox PARC Technical report CSL-85-1.