Research into the Design of Distributed Directory Services

by Steven David Benford, BSc. (Hons)

Thesis submitted to the University of Nottingham
for the degree of Doctor of Philosophy, October 1988.

I would like to thank my supervisor, Mr. Hugh Smith, for the valuable guidance he has provided throughout my career as a postgraduate student. His advice, comments and support have been essential to the production of this thesis. In addition, I would like to thank him for encouraging my work in other areas and enabling its publication and presentation.

I am also very grateful to Steve Kille, Julian Onions and Karl-Heinz Weiss for their advice throughout my research. In addition, I would like to thank Penny Jewitt, Andy Cheese, Peter Cowan and Graeme Lunt for their assistance in the production of this thesis. Finally, I would like to thank all of my colleagues within the Computer Science Department for their friendship over the past years.

This thesis is dedicated to my mother and my father for all the support and encouragement they have given me.

Contents

CHAPTER 2

A LAYERED DIRECTORY ARCHITECTURE

CHAPTER 3

THE GLOBAL CONCEPTUAL LAYER

INFORMATION MODEL AND ABSTRACT OPERATIONS

CHAPTER 4

THE MANAGEMENT OF DIRECTORY INFORMATION

CHAPTER 7
THE LOCAL INTERNAL LAYER
A DIRECTORY IMPLEMENTATION

List of Figures

Abstract

Distributed, computer based communication is becoming established within many working environments. Furthermore, the near future is likely to see an increase in the scale, complexity and usage of telecommunications services and distributed applications. As a result, there is a critical need for a global Directory service to store and manage communication information and therefore support the emerging world-wide telecommunications environment.

This thesis describes research into the design of distributed Directory services. It addresses a number of Directory issues ranging from the abstract structure of information to the concrete implementation of a prototype system. In particular, it examines a number of management related issues concerning the management of communication information and the management of the Directory service itself.

The following work develops models describing different aspects of Directory services. These include *data access control* and *data integrity control* models concerning the abstract structure and management of information as well as *knowledge management, distributed operation* and *replication* models concerning the realisation of the Directory as a distributed system.

In order to clarify the relationships between these models, a layered directory architecture is proposed. This architecture provides a framework for the discussion of directory issues and defines the overall structure of this thesis.

This thesis also describes the implementation of a prototype Directory service, supported by software tools typical of those currently available within many environments. It should be noted that, although this thesis emphasises the design of abstract directory models, development of the prototype consumed a large amount of time and effort and prototyping activities accounted for a substantial portion of this research.

Finally, this thesis reaches a number of conclusions which are applied to the emerging ISO/CCITT X.500 standard for Directory services, resulting in possible input for the 1988-92 study period.

Abstract

Distributed, computer based communication is becoming established within many working environments. Furthermore, the near future is likely to see an increase in the scale, complexity and usage of telecommunications services and distributed applications. As a result, there is a critical need for a global Directory service to store and manage communication information and therefore support the emerging world-wide telecommunications environment.

This thesis describes research into the design of distributed Directory services. It addresses a number of Directory issues ranging from the abstract structure of information to the concrete implementation of a prototype system. In particular, it examines a number of management related issues concerning the management of communication information and the management of the Directory service itself.

The following work develops models describing different aspects of Directory services. These include *data access control* and *data integrity control* models concerning the abstract structure and management of information as well as *knowledge management*, *distributed operation* and *replication* models concerning the realisation of the Directory as a distributed system.

In order to clarify the relationships between these models, a layered directory architecture is proposed. This architecture provides a framework for the discussion of directory issues and defines the overall structure of this thesis.

This thesis also describes the implementation of a prototype Directory service, supported by software tools typical of those currently available within many environments. It should be noted that, although this thesis emphasises the design of abstract directory models, development of the prototype consumed a large amount of time and effort and prototyping activities accounted for a substantial portion of this research.

Finally, this thesis reaches a number of conclusions which are applied to the emerging ISO/CCITT X.500 standard for Directory services, resulting in possible input for the 1988-92 study period.

Preface

The goals of this thesis

This thesis describes the specification and implementation of a prototype *Directory Service* supporting computer based communication within an *Open Systems* environment. The Directory service is a specialised, globally distributed, database providing humans and applications with the information required in order to communicate. In particular, it manages a distributed, global name space for communication entities and supports the management of communication information shared between many cooperating organisations and services.

This thesis explores a number of *management* issues relevant to the Directory service. These issues concern the management of directory information and the management of the Directory service itself.

The need for a global Directory service has been apparent within the electronic mail community for some time and is becoming critical as the community continues to expand rapidly [SIRB84]. In addition, a Directory service is required to support other present day applications such as *file transfer*, *remote login* and *remote job execution* [SANT86]. Support is also required for emerging applications in the area of *Computer Supported Cooperative Work* (CSCW) such as *group communication services* and *distributed office systems* [WILS88] as well as for many new applications in the future.

This obvious need for Directory services has been recognised by the *International Standards Organisation* (ISO) and *International Telegraph and Telephone Consultative Committee* (CCITT) standardisation bodies. As a result, 1988 will see the introduction of the first joint ISO/CCITT *X.500* standard for Directory services [CCITT-X500]. This standard should provide a basic service, sufficient to plug the current gap in communication requirements. However, many unresolved issues, particularly management issues, mean that 1988 X.500 is unlikely to cope with a growing number of users and applications in the future. Thus, 1988 marks a watershed for the study of Directory services when one can look back on the culmination of previous work, in the form of 1988 X.500, and also look forward to those issues requiring solution as the standardisation process enters the 1988-1992 study period. The publication of this thesis is therefore particularly relevant at the present time.

The urgent need for a global Directory service has provided the motivation for this research. The overall goals of this work are defined below:

1 The first goal is the specification of a distributed Directory service including mechanisms supporting the management of information and the management of the service itself. This requires the development of abstract models describing issues such as *access control*, *data integrity*, *directory configuration* and *replication*.

2 The second goal is the implementation of a prototype Directory service using these models. Prototyping is an integral part of the overall design process and aims to refine the results of specification work. Furthermore, the prototype explores the use of several software tools, typical of those available within many organisations today, for constructing Directory services.

It is important to note that this research has occurred in parallel with the development of X.500 during the years 1985-1988. This thesis assumes several past and present X.500 concepts and models as the basis for new ideas. However, this research has taken a different slant to X.500, adopting a more database oriented approach concentrating on management and distribution issues beyond the scope of the standard. In contrast, the ISO and CCITT work has required the specification of an immediately implementable model, resulting in many of these more complex issues being postponed for future study.

The structure of this thesis

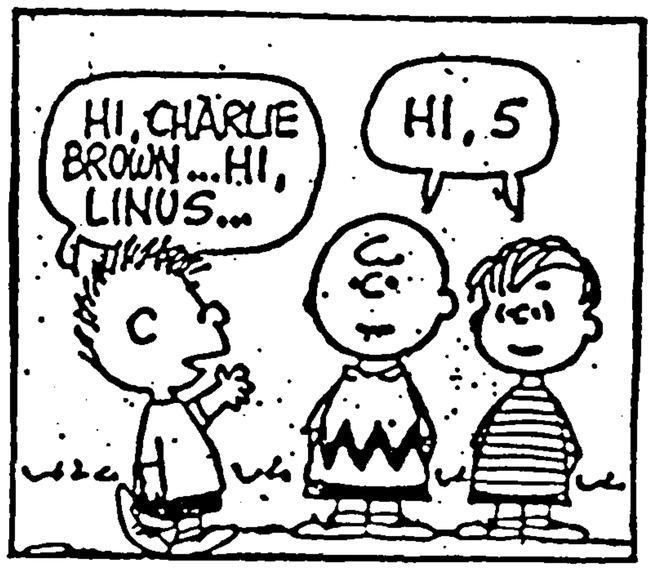
The goals of this thesis identify the need for both specification and implementation work within my research. Specification work develops a number of abstract models and ideas whereas implementation work concerns the realisation of these ideas using specific tools and systems. In order to clearly present both these areas of work while preserving the separation between the abstract specification and the concrete implementation, a *layered directory architecture* is used to structure this thesis. The proposed architecture supports several directory models at different levels of abstraction allowing a clear separation of issues. Consequently, the structure of this thesis is not based on the chronological progression of my research, where specification and implementation occurred in parallel, but is based on the logical top-down structure of the layered architecture. I believe that this facilitates a more coherent presentation.

The layered architecture views the Directory as a distributed database and is derived from general database architectures including the ANSI/SPARC centralised database architecture [ANSI75] and the distributed architecture of Dean et al as used in the PRECI* distributed database system [DEEN82]. Each layer of the architecture describes issues at a different level of abstraction. Layers are therefore, generally, independent with higher layers representing abstract user and application views of information and lower layers being concerned with distribution and storage issues.

The *top-down* approach adopted by this thesis begins by examining issues belonging to higher layers, such as the information model, access controls and data integrity, and proceeds to discuss lower layer issues, such as the management of distribution, replication and finally the implementation of a prototype using a number of specific tools. Thus, management issues are resolved within the early chapters and implementation work is left to the later chapters.

The layered directory architecture and its relationship to subsequent chapters are described in chapter 2. However, the following chapter descriptions should be sufficient to familiarise the reader with the general structure of this thesis for the time being.

- Chapter 1 examines the requirements of the Directory service and reviews a number of existing *nameservers*. It also provides an introduction to naming issues and an overview of the 1988 X.500 Directory standard.
- Chapter 2 develops the specific layered directory architecture and explains its role in structuring this thesis.
- Chapter 3 specifies the directory abstract information model and operations. These form the basis of all later work.
- Chapter 4 examines the management of directory information and extends the basic information model to include data access control and data integrity mechanisms.
- Chapter 5 discusses the distributed operation of the Directory service and specifies the distribution of the abstract information model between a set of server applications called *Directory System Agents*. It also develops a number of system management mechanisms supporting navigation and distributed operations following system reconfiguration.
- Chapter 6 concerns the use and management of *replicated* information within the distributed Directory system. It considers the management of replication and its impact on the work of chapter 5.
- Chapter 7 describes a pilot directory implementation. It outlines the overall structure of the implementation and the use of the *RTI Ingres* Relational Database Mangement System and *ISODE* software tools in its construction. Although implementation work is only considered within this chapter, the development of the prototype accounted for a substantial part of the research leading to this thesis.
- Chapter 8 presents the conclusions of this thesis. In addition, it considers the implications of this work for the future development of X.500 and examines the role of Directories within a future general distributed management framework.


HI, CHARLIE BROWN... HI, LINUS...
HI, 5


IF WE ALL HAD NUMBERS INSTEAD OF NAMES, WHAT NUMBER DO YOU THINK YOOD LIKE TO HAVE?
HOW ABOUT 3.1416?


I DON'T KNOW... I HAVE A FEELING THAT EVERY TOM, DICK AND HARRY WOULD BE CALLED 3.1416!

Chapter 1

Introduction

This chapter provides the background to this thesis and aims to familiarise the reader with the general concepts and ideas emerging from previous work in this area.

- Section 1.1 considers the role of the Directory service in supporting computer based communication. In particular, it examines the characteristics of communication within an Open Systems environment and demonstrates the need for a distributed Directory service to store and manage the *communication information* required by cooperating distributed applications and their users.
- Section 1.2 places this research into historical perspective by reviewing existing systems partially fulfilling the role of Directory services. It analyses their strengths and weaknesses and indicates their limitations, particularly the lack of support for information and system management.
- Section 1.3 provides an introduction to the critical issue of distributed naming and reviews previous work in this area. It discusses human oriented naming, compares *absolute* and *relative* naming schemes and concludes by describing *hierarchical* naming, used as a basis for the Directory service.
- Section 1.4 reviews the 1988 X.500 standard and informally introduces some of the basic terminology and models adopted by this research. In addition, it lists some fundamental limitations of the standard.
- Section 1.5 clarifies the relationship between this thesis and the X.500 standard. In particular, it indicates where current and previous X.500 concepts have been used as a basis for this research.

1.1. The need for Directory services within the OSI environment

Successful communication is a fundamental requirement for the achievement of human tasks and goals. The availability of computer networks has opened the door on a new era where humans use computers to communicate in ways not previously possible. Electronic communication via computer networks is well established and accepted within the computer

science academic environment and many institutions support electronic mail, news and conferencing facilities. This usage is being extended to the business environment where, along with document creation and storage facilities, computer based communication will constitute a major part of the office systems of the future [PRIN87].

Humans require information before communication can occur. They need to know the names and capabilities of the entities with whom they wish to communicate. They also wish to locate services to help achieve tasks. For example, they wish to know the name of a new contact within another organisation, the names of bulletin boards on a certain subject or the joining procedure for a specific distribution list. Applications also require information if they are to provide communication services. In particular, they need to map the names of entities onto application specific addresses.

The Directory Service is a globally distributed database for telecommunications services fulfilling two main roles in supporting communication:

- It acts as an *information provider*, allowing both humans and applications to access the information required for communication.
- It acts as an *information manager*, maintaining communication information in the face of changes to users, networks, organisations and services.

The following sections observe trends within computer based communication (section 1.1.1) and explain the need for a Directory service to play the two roles described above (sections 1.1.2 and 1.1.3).

1.1.1. Trends within computer based communication

An increasing number of people are using distributed computer services to communicate and achieve tasks within their working environment. The following are examples of user activities involving distributed computer services.

- Inter-personal communication via electronic mail.
- Subscription to a bulletin board or news service.
- Participation in a real time conference.

Each of these activities might involve the use of several cooperating distributed services or applications. For example, subscription to a bulletin board might employ the electronic mail service to carry new messages to the board, involve a document storage service in maintaining the messages on the board and might employ an authentication service to verify the identity of subscribers.

The following are examples of commonly available services*:

Examples of today's communication services	
Generic service	Examples
Electronic mail	X.400, RFC-822
File transfer	FTAM, Arpanet FTP
Document storage	ECMA FRS
Remote login	Telnet, X.29
Remote job execution	JTMP
Real time conferencing/conversation	UNIX Talk/Write
News/Bulletin Board	USENET News

In addition to the services available today, several research groups are studying mechanisms for integrating individual services into office systems environments and group communication frameworks [ISO-DOA88, SMIT88, PANK87, WILB88]. In fact, a general model of future communication is evolving where humans achieve tasks via the structured coordination of many communication services within their distributed environment. This work is closely related to the topic of *Computer Supported Cooperative Work* (CSCW) [CSCW86, WILS88]. One example group communication architecture, proposed by Bogen and Weiss, illustrates the point. Under this model, users are supported by many services which, in turn, are themselves supported by other services [BOG88].

The number and variety of communications services is also increasing and this trend is likely to accelerate with the introduction of new technologies such as ISDN [END88].

Not only are the number and complexity of communications services increasing, but network communities are rapidly expanding and communication already occurs on a global scale. The number of communicating entities may reach billions as is presently the case with the telephone network and communication will span international, cultural and organisational barriers.

Much future communication may occur within an *Open Systems* framework allowing the interconnection of information processing systems of different makes, sizes and ages showing a high degree of autonomy in operation and management [ISO-OSI84]. The Open Systems philosophy allows autonomous organisations to co-exist and cooperate in providing network services. Thus, the different services employed by a user might operate under different managements, use different software and run on a variety of hardware.

---

*References are included in the bibliography.

In summary, one can observe the following trends in computer based communication:

- The size of communities is very large and expanding.
- Communication occurs on a global scale spanning international and cultural barriers.
- Tasks may be achieved via the structured coordination of many cooperating services.
- The "users" of a service might be humans or other applications acting on the behalf of humans.
- The number and variety of services and service providers is increasing.
- Communication often occurs within an Open Systems framework of different machines and applications under autonomous management.

1.1.2. The information required for communication

Communication using distributed services requires information concerning the names, locations, environments and capabilities of communicating entities and services. The volume and diversity of this information will be very large for communication in the environment described above. This point is illustrated by considering the information required to use the telephone service and extending to the case where a number of more complex services are supporting communication tasks.

The information required to use the telephone service

The telephone service is the best established telecommunication service available today and has billions of subscribers distributed throughout the world. If one considers the use of the telephone service, one observes the following requirements for information:

- The initiator of a telephone call must know the telephone number of the intended recipient. In many cases, this number is not known to the initiator and can be obtained from an information service (the *White Pages Telephone Directory*). The information service requires that the initiator identify the recipient by supplying enough information to distinguish them from all other possible recipients.
- The telephone network requires information in order to provide the service. It needs to know the telephone numbers of callers so that the network *circuit switches* can establish a connection between them.
- Use of the telephone service increases if subscribers are able to advertise themselves. Users of the telephone service can query the *Yellow Pages Directory* to obtain the telephone numbers of subscribers grouped under generic descriptions (e.g. *Plumbers, Electricians* ..). This generic information is not strictly necessary for the telephone

service to function but clearly encourages use of the service and allows users to identify the people with whom they wish to communicate. The yellow pages service benefits both the users of the telephone service and the providers of the telephone service.

The information required for computer based communication

The following paragraphs abstract these observations and apply them to general computer based communication as described above. The following are generally true:

- The user of a service requires information necessary to communicate via the service. The names of other users are examples of this type of information.
- The services themselves require information in order to function. The addresses (network locations) of subscribers are examples of this type of information.
- Information advertising the capabilities of the service and listing subscribers to the service will increase its usage and benefit both service providers and users.

For example, use of a distribution list requires that subscribers know of its existence, know the address of the list and understand its purpose. The application implementing the distribution list requires information to determine whether an entity is allowed to use the list and to determine the names and addresses of the list members for distribution purposes. The managers of the list will wish to advertise its presence to the intended audience and furthermore, information describing the purpose of the list will encourage its correct usage and therefore benefit the subscribers.

The diversity and volume of the information required for future communication within an Open Systems environment can be expected to be far greater than for the telephone service alone. This is because of the greater number and complexity of the services involved. The following paragraphs demonstrate the nature and diversity of this *communication information*.

A *communication entity* can be thought of as any entity involved in a communication process. This might include humans, application entities, groups and services. The table below gives a few examples of communication entities and their communication information

Communication entities and information	
Entity	Information
persons	names, addresses, titles, responsibilities
groups of persons	names, addresses, members
hosts	names, addresses, capabilities
roles	names, conventions
organisations	names, addresses, descriptions
mailboxes	addresses, capabilities

For example, a distribution list might be represented by a group of entities associated with the following information, required by a mail or bulletin board service:

Example distribution list information
Name of the group
Names of members of the group
Description of the purpose of the group
Name of the person maintaining membership of the group

In its role as an information provider, the Directory service must provide users with information concerning the above kinds of communication entities.

Naming

Fundamental to the role of the Directory as an information provider, is the issue of naming [WHIT84, SHOC78].

The name of each communication entity is of major importance. Names allow the identification of entities and provide the basic handle to access their information. Without the names of communication partners, communication would be impossible. For example, sending an inter-personal message requires the name of the recipient. Subscribing to a bulletin board requires the name of the particular board. Consequently, the provision of a global communication entity name space is a primary goal of the Directory service. This is discussed in section 1.3.

In summary, computer based communication requires information to ensure that users can use the service, that the service can function and to advertise the service and its subscribers thus increasing use of the service.

There will be a large volume and diversity of this information due to the existence of many different services having complex functionality. Furthermore, services will have a large number of subscribers. One role of the Directory service is to provide its users with this

information. In particular, this involves the provision of a global name space for communication entities.

1.1.3. The management of information

The second major role of the Directory service concerns the *management* of communication information.

Communication information reflects the state of real world communication entities and is vital to successful communication as described above. The management of communication information describes its consistent update to reflect changes to real world entities. For example, changing the address of a host, the name of a person or the administrator of a distribution list. Without its correct, consistent update, this information would soon become invalid and communication would break down.

To understand the nature of the management problem, consider current approaches to the storage and management of communication information. At the present time, the information utilised by a service is usually stored in a specific service information base. Individual information bases vary in size, distribution and complexity and are usually maintained on an organisational basis by a small group of administrators. For example, at Nottingham, the *mmdf* mail service [KING84] relies on routing tables, maintained in individual message transfer agents, and the *remote login* service [DDN-TELNET] utilises host information stored in local textfiles.

There are many problems with this approach to the maintenance of communication information. One problem concerns the general lack of support for information management within the individual information bases themselves. Solutions based on editing textfiles and tables exhibit the following drawbacks:

- They do not *scale* to large volumes of information.
- They do not provide access control and data integrity mechanisms, recognised as essential for supporting the correct update of information within general purpose database systems [DATE77].

A second major problem concerns the lack of *cooperation* between different service information bases. In an environment where many services cooperate to achieve tasks, a single update to information may affect many information bases. For example, changing a person's name may affect a mail service, bulletin board service and real-time conference service. The lack of cooperation between information bases results in the following specific problems:

- Services do not share common information and cooperation is therefore impeded.

- There is no mechanism ensuring that an update to one information base is consistently propagated to all other affected information bases.
- Each information base supports its own access protocol with similar functions being reproduced many times. This is a waste of effort.

The classes of problem described above may be summarised by the phrases: *lack of scale* and *lack of coordination*. These issues will become critical as the size of the user community and number of services grows.

The Directory service provides the solution to these problems by maintaining a globally unified information base accessed by many services. The management of communication information within a single Directory service solves the problem of *coordination*. Furthermore, the Directory should be designed to operate on a *global scale*. In order to facilitate the global management of communication information, the Directory should support the following:

- A data access control mechanism controlling the legality of updates.
- A data integrity mechanism controlling the validity of updates.
- The definition of *management policies* reflecting real world policies.

Support for the management of information is a major goal of this thesis and these issues will be defined and discussed in later chapters.

In summary, the consistent management of communication information is vital to the operation of communication services. At the present time, management is typically achieved by ad-hoc methods within a variety of service information bases. This results in problems of *scale* and *coordination*.

A major role of the Directory service is to unify these discrete service information bases into a single, global *Directory Information Base*. This requires mechanisms supporting the consistent management of communication information on a global scale.

The previous sections have described two major roles of the Directory service in supporting computer based communication. The first is that of an information provider, allowing communication entities to retrieve the information necessary to establish and support communication. The second is that of an information manager, facilitating the distributed management of communication information on a global scale. The following section demonstrates that previous work in this area generally ignores this second role. Consequently, the management of communication information will form a major aspect of this thesis.

1.2. A review of existing systems

The need for a global Directory service has been recognised for several years, particularly within the electronic mail community where *user friendly naming* and the management of name spaces have been important issues [WHIT84, SIRB84]. The development of electronic mail has required the evolution of naming techniques and the expansion of naming schemes to global proportions. The growth of naming schemes has encouraged the development of simple Directory systems called *nameservers* which, as the name implies, are primarily concerned with the naming (and addressing) of mail users and other mail entities in a distributed environment. The following list of nameservers and Directory servers includes many still in use today.

Nameservers
Grapevine [BIR81]
Clearinghouse [OPP81]
BIND [TER84a]
The CSNET Nameserver [LAN83]
The NRS [LAR82a]
ECMA TR-32* [ECMA-TR32]
Thorn [KIL87a]
QUIPU [KIL88b, KIL88c]
Hesiod [DYER87]

The following sections briefly describe and compare three of the above nameservers and conclude by drawing together their common and important features. The three nameservers are the *Clearinghouse*, *BIND* and the *CSNET Nameserver*. Although intended to solve the naming problem within different environments, these three exhibit several common features as described below.

The purpose of these reviews is to to place this research in a familiar context and introduce the reader to fundamental concepts. Terminology will be formally defined in later chapters.

1.2.1. The Clearinghouse

The *Clearinghouse* nameserver was developed by the Xerox corporation in the early 1980s to solve the problem of naming and locating *objects* in a distributed environment [OPP81]. The nameserver is intended to support several applications, including electronic mail, and

*This is really a specification of which there may be many implementations.

has to deal with a variety of information concerning different types of object. Examples of objects are *machines*, *workstations*, *fileservers* and *people* as well as groups of these, represented by *distribution lists*.

Each object has a *name*, distinguishing it from all other objects, which may be used to access the information stored about the object. Names are distinct from *addresses* describing the physical location of objects. The name to address mapping is a fundamental Clearinghouse service. However, the Clearinghouse also allows objects to be located and accessed via generic groupings such as *printer* or *workstation*. The service is implemented by a set of physically distributed servers and the Clearinghouse design specifies methods of locating and replicating information in a globally distributed environment. The following sections describe specific aspects of the Clearinghouse design in greater detail.

Naming

All Clearinghouse objects are named under the same convention regardless of type and thus share a common namespace. The namespace describes a three level hierarchy where the world of objects is divided into *organisations* and then subdivided into *domains* and finally *local names*. These divisions are logical and do not reflect the physical or geographical locations of objects. Each object has a *distinguished name* consisting of a character string of the form *L@D@O*, where *L* is the local name, *D* the domain name and *O* the organisation name. An example name space is shown in figure 1.1 below. Distinguished names are unique and unambiguous. This means that each object has exactly one distinguished name and each distinguished name describes exactly one object. In addition to its distinguished name, an object may have one or more *aliases* providing alternative, but still unambiguous, names for the object. Aliases are syntactically identical to distinguished names.

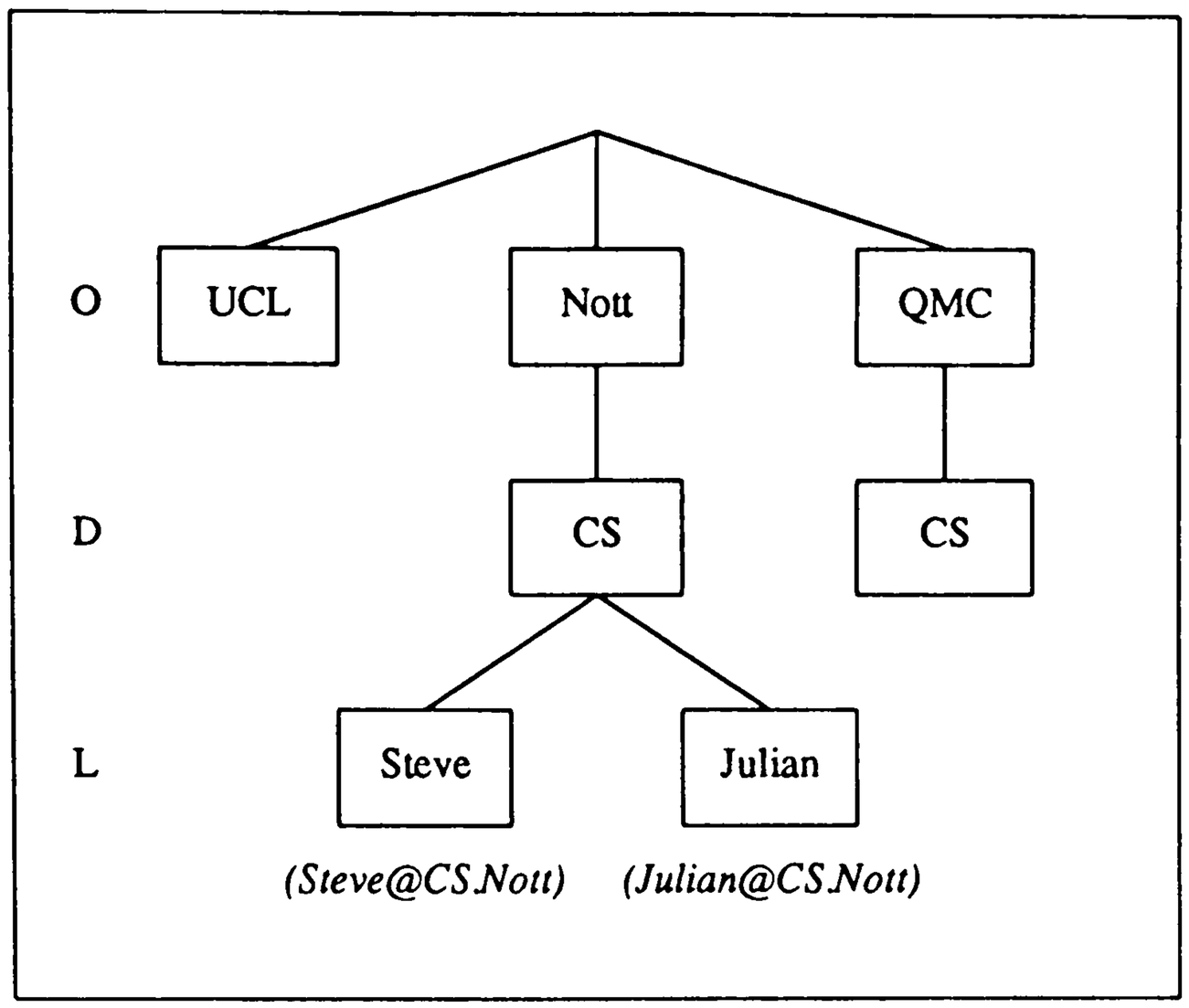


Figure 1.1: An example Clearinghouse namespace.

The Clearinghouse is capable of locating an object given its distinguished name, one of its aliases or a partial name match which can be expanded to a full three level name.

Properties and Operations

In addition to its name, a Clearinghouse object contains a set of *properties* representing its physical characteristics such as addresses, capabilities, passwords and descriptions. A property is an ordered tuple consisting of a *name*, *type* and *value*. For example, a "person" object might have the property, <*Password*, *individual*, *bananas*> representing their real world password. There are two possible values of the property type: *individual*, indicating that the property value is atomic (i.e. an uninterpreted block of data), or *group*, indicating that the value is to be interpreted as a set of names. An object may possess an arbitrary number of properties including several properties with the same name. The following example shows a possible "person" object.

```
Steve@CS@Nottingham -> {
        <Title, individual, "postgraduate student">
        <Password, individual, "bananas">
        <File server name, individual ,"sheriff@CS@Nottingham">
        <Printer names, group, "beth@CS@Nottingham","anadex@CS@Nottingham"> }
```

The basic functionality of the Clearinghouse is to map names to sets of properties. In addition, the Clearinghouse allows clients to create and manipulate names and aliases, retrieve and manipulate individual properties of a named object and manipulate names in group properties. *Set operations* allow a client to determine whether a name belongs to a set of names and to add and delete themselves (or other names) to and from sets.

The Clearinghouse provides *searching* facilities allowing a client to retrieve the names of those objects containing specified properties. Searching implements a *set of properties to set of names* mapping, loosely described as *generic naming* (i.e. naming generic sets of objects by specifying common properties).

Enumerate operations allow clients to explore the name space by returning the names of all objects belong to a named domain or all domains belonging to an organisation.

Access Control

The functionality described above is supported by the Clearinghouse *access control mechanism* governing which users may perform which operations. This mechanism preserves the integrity of information by ensuring that all updates are legal and that only permitted clients can perform operations. The access control mechanism recognises two classes of user: *domain system administrators* and *general users*. Administrators have the ability to explore the name space, creating and deleting new names, aliases and properties. General users have more restricted abilities to read and manipulate individual properties and members of sets of names.

The access control mechanism is implemented by *Access Control Lists* (ACLs) stored within the properties of objects. An ACL associates a set of names with some named operations. It is interpreted as giving permission for any of the named objects (typically persons) to perform any of the specified operations on the property.

Distributed Operation

The Clearinghouse name space is partitioned among a set of physically distributed server applications known as *clearinghouse servers*. There are three types of clearinghouse server called *organisation, domain* and *local* servers. These are responsible for subsets of organisation, domain and local names respectively.

A clearinghouse server is able to perform those operations accessing names for which it is responsible. In addition, it may supply clients with the names of more responsible clearinghouse servers. The routing of operations requires that servers know of each others existence and responsibilities. Clearinghouse servers are arranged so that they can always route

queries hierarchically. This is possible if each server knows the name of its parent and each domain or organisation server knows the responsibilities of all its siblings. This arrangement means that a client need contact a maximum of four Clearinghouse servers during the execution of an operation.

More than one clearinghouse server may be responsible for an object, leading to the possibility of conflicting updates when the name or properties of an object are simultaneously modified at different servers. The Clearinghouse supports an update mechanism resolving conflicts by the use of *timestamps*. This mechanism guarantees to bring eventual consistency to Clearinghouse information although *transient inconsistency*, where different versions of an object temporarily exist at different servers, is possible and is considered acceptable.

1.2.2. The Berkeley Internet Name Domain Server (BIND)

The *Berkeley Internet Name Domain Server (BIND Nameserver)* was developed at Berkeley, University of California, to provide a uniform means of naming and locating resources in the UNIX† internet community with the aim of providing a *more transparent and less troublesome* computing environment [TER84a]. The protocols and information structure utilised by the BIND nameserver are specified in the *RFC 882* [MOCK83a] and *RFC 883* [MOCK83b] Request For Comment documents and the BIND software has been widely adopted throughout the Arpanet to implement a large scale distributed name service. Due to the size and age of the Arpanet, BIND is perhaps the most established and tested nameserver available. In addition, several projects have considered its extension to support greater functionality [DYER87].

The resources managed by BIND include named objects such as hosts, user mailboxes and server ports occupying a common, hierarchical namespace. Users are provided with operations to interrogate the name service. These support the mapping of names to properties and also the completion of partially specified names. Some versions of BIND support operations for the remote maintenance of information. However, this is generally left as a local matter. BIND does not support a dynamic, user level access control mechanism.

Responsibility for the management of the namespace is divided between a set of distributed server entities called *nameservers*. These cooperate to resolve queries and provide the user service. Like the Clearinghouse, BIND is therefore a distributed system.

The following sections describe specific aspects of the BIND nameserver in greater detail.

---

† UNIX is a trademark of Bell Laboratories.

Naming

The BIND nameserver maintains a hierarchical, tree structured name space. A node of the tree represents a *domain* responsible for naming its immediate children (sub-domains). Leaf nodes of the tree represent resources as described above. Each node is identified by a label and the name of a domain is therefore the concatenation of the domain labels from the root of the tree to the named domain. These labels are written from right to left and separated by dots. Labels must be unique within the same domain ensuring that each name unambiguously denotes just one node of the naming tree.

Unlike the Clearinghouse which supports a three level naming hierarchy, BIND allows its naming tree to be of unlimited depth. This is shown in figure 1.2.

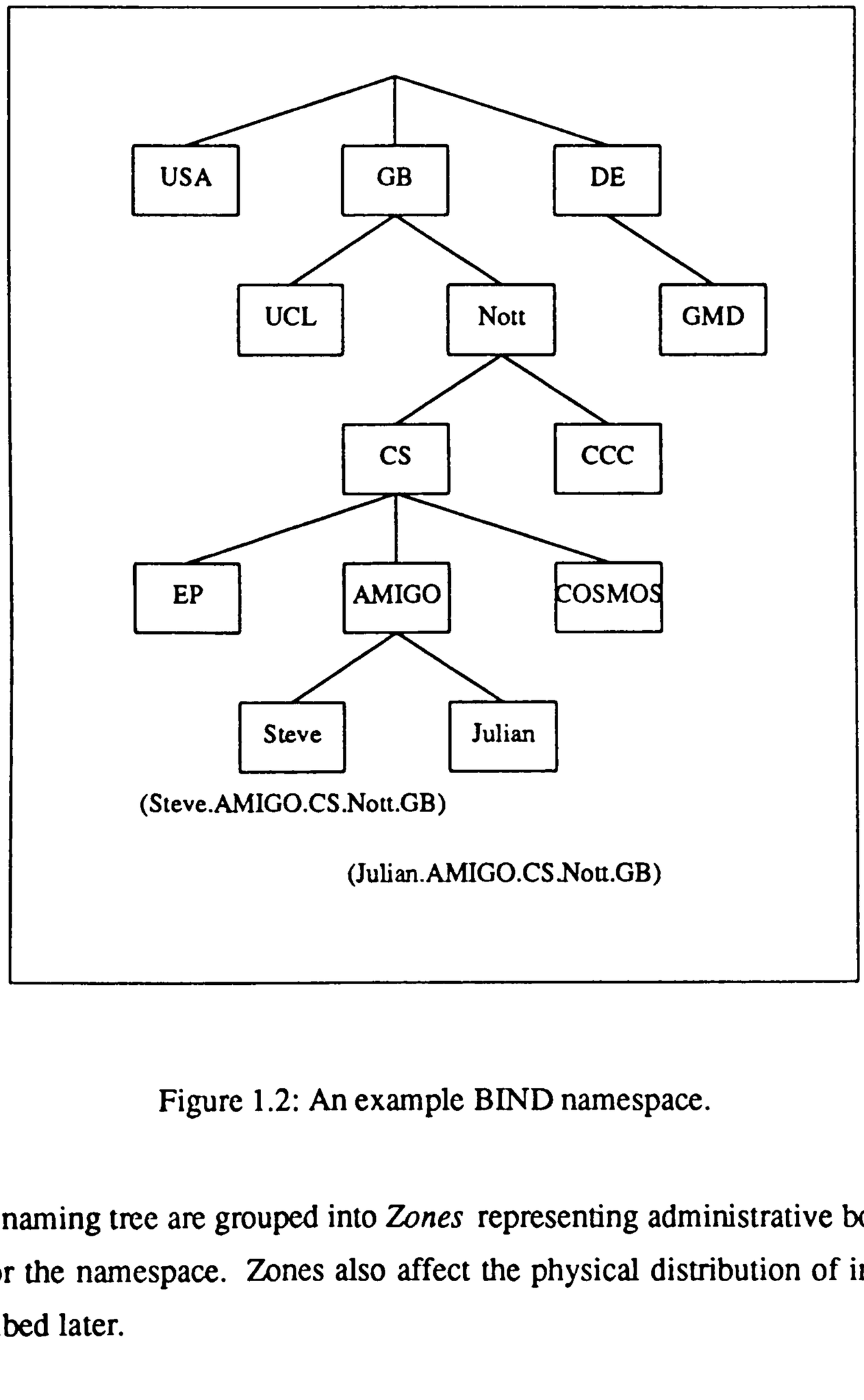


Figure 1.2: An example BIND namespace.

Nodes of the naming tree are grouped into *Zones* representing administrative boundaries and authorities for the namespace. Zones also affect the physical distribution of information as will be described later.

BIND supports a basic name to resource mapping. In addition, some versions include the functionality to complete partially specified names and map resource information to names (*Inverse Queries*) although this is not generally supported.

Resource records and Queries

The characteristics of a resource are represented by a set of *resource records* fulfilling the role of Clearinghouse *properties*. A resource record has the following structure:

Figure 1.3: Structure of a BIND resource record

The *owner* specifies the name of the resource to which the record belongs. The *type* specifies the generic type of information represented by the resource record. For example, *host address (A), mail destination (MD)* or *authoritative name server (NS).* The *class* specifies which of two formats the record takes. The possible formats are *Arpa Internet (IN)* and *Computer Science Network (CSNET).* Class is a historical anomaly supporting the different information syntaxes of the two major networks served by BIND. The *value* represents the value of this information type for this specific resource. In addition to the above, each resource record may be associated with information specifying its length and "time to live". The following example shows a set of resource records associated with a specific resource from figure 1.2.

BIND resource records			
CS.NOTT.GB	MD	IN	CS.NOTT.GB
CS.NOTT.GB	MF	IN	COSMOS.CS.NOTT.GB
CS.NOTT.GB	A	IN	10.1.0.32

The most advanced implementations of BIND support three types of query for the retrieval of resource information.

- A standard query allows a user to specify the owner, type and class of some resource records and returns those records matching the query. This corresponds to a name to set of properties mapping.

- An inverse query allows the user to specify a resource record and returns the names of those resources containing the record. This corresponds to a property to name mapping, sometimes referred to as *generic naming*.
- A completion query allows a user to specify a partial domain name and returns the set of resource records matching the completed name. Completion queries provide users with the ability to search the name tree.

Some versions of BIND support operations facilitating the update of information. However, most versions maintain information via local protocols accessing text files of resource records at a given host. These files are loaded into a nameserver during initialisation.

Distributed operation

The realisation of the BIND name service is divided between two classes of entity:

- A *nameserver* is a database application at a host, responsible for a portion of the name space.
- A *resolver* presents the BIND interface to a user and manages their dialogue with the name service.

Thus, the complete BIND service is realised as a set of nameservers, collectively storing the entire naming tree, and a set of resolvers representing BIND clients.

The name tree is divided into zones, distributed between nameservers so that each name server is *authoritative* (responsible) for one or more zones. In addition, a zone can be held by more than one nameserver.

There is one *primary name server* maintaining the master copy of each zone. All other name servers containing the zone are *secondary name servers* for that zone (they may be primary servers for other zones). The division between primary and secondary name servers supports a *single master update* procedure for the information belonging to each zone.

Name servers are organised hierarchically so that a name server, not authoratitive for a query, may use the hierarchy to return the name and address of another name server which might be authoratitive. Thus, queries are navigated in a distributed fashion.

A resolver is responsible for managing the distributed navigation of a query until it reaches an authoratitive nameserver. Resolvers manage navigation via several name servers which either return the results of the query or the name of another nameserver. Thus nameservers do not interact directly to resolve user queries. However, they might interact to circulate updated zone information. The interaction between resolvers and nameservers is shown in figure 1.4.

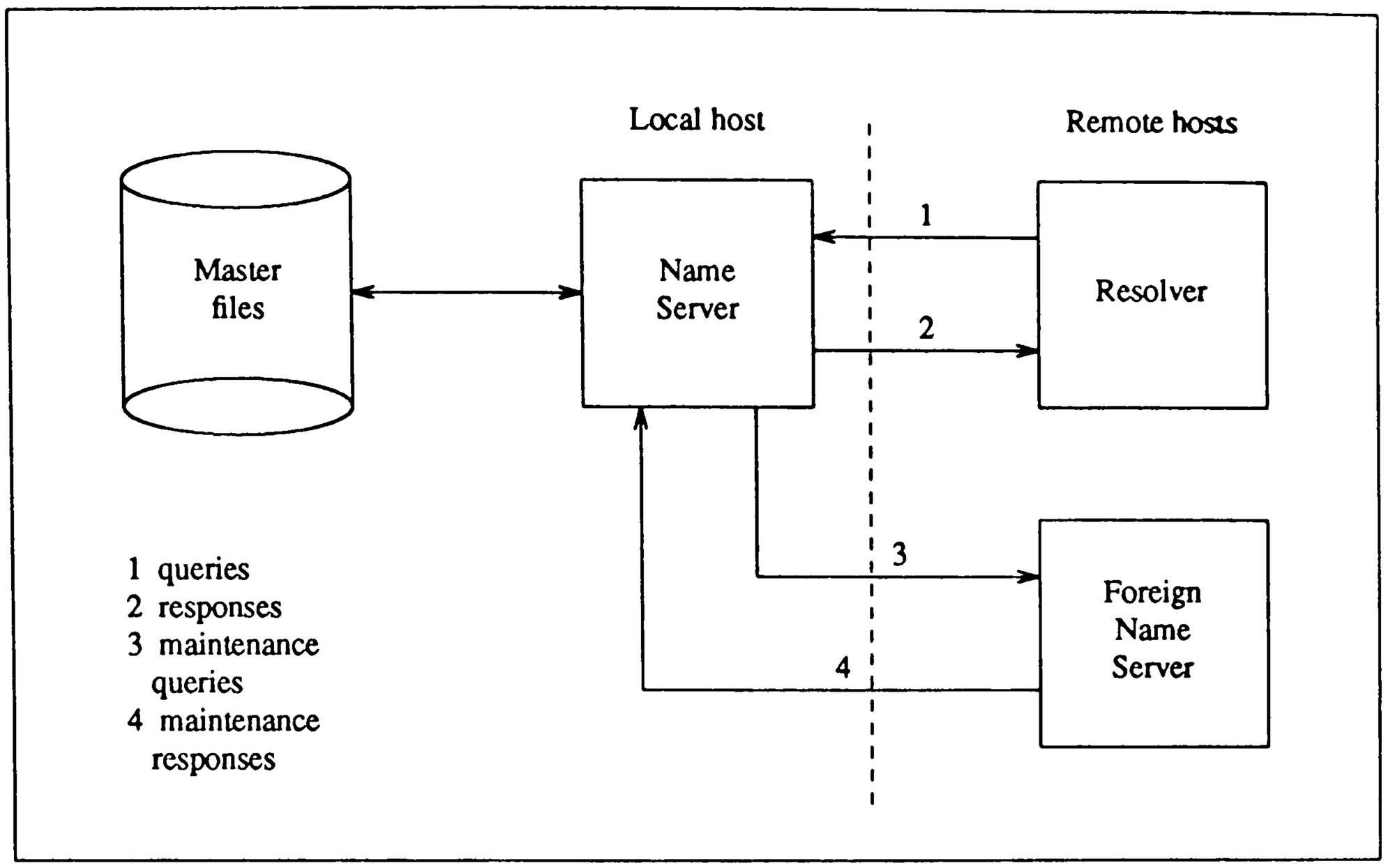


Figure 1.4: Interaction between BIND resolvers and nameservers.

1.2.3. The CSNET Nameserver

The CSNET has been developed to provide network services to research groups throughout the USA. It allows users to connect to the *Arpanet, Telenet* or *Phonenet* networks and communicate via the US Department of Defence *TCP* and *IP* standard protocols [DDN-TCP]. The CSNET Nameserver has been developed at the university of Wisconsin to aid users in locating resources and sending electronic mail via the CSNET [LAN83]. The use of different naming and addressing conventions in each of the CSNET's constituent networks is a major hurdle to effective communication and the CSNET Nameserver is intended to free users from the complexities of mail addressing under these different schemes.

The CSNET Nameserver provides a *registry* of information holding entries for all registered CSNET users. These entries contain descriptions of user's names, addresses and a number of descriptive keywords such as mail addresses, phone numbers and passwords. The registry is stored on a single host computer and interacts with users via an *agent* program resident at their local host. Thus, unlike both BIND and the Clearinghouse, the CSNET Nameserver is a *centralised system.*

Each agent manages dialogue with the registry via a remote access protocol and provides the user with additional functionality such as a local *nickname* space. Users may also interact with the central registry via the electronic mail service. The centralised architecture of the CSNET nameserver is shown in figure 1.5 below.

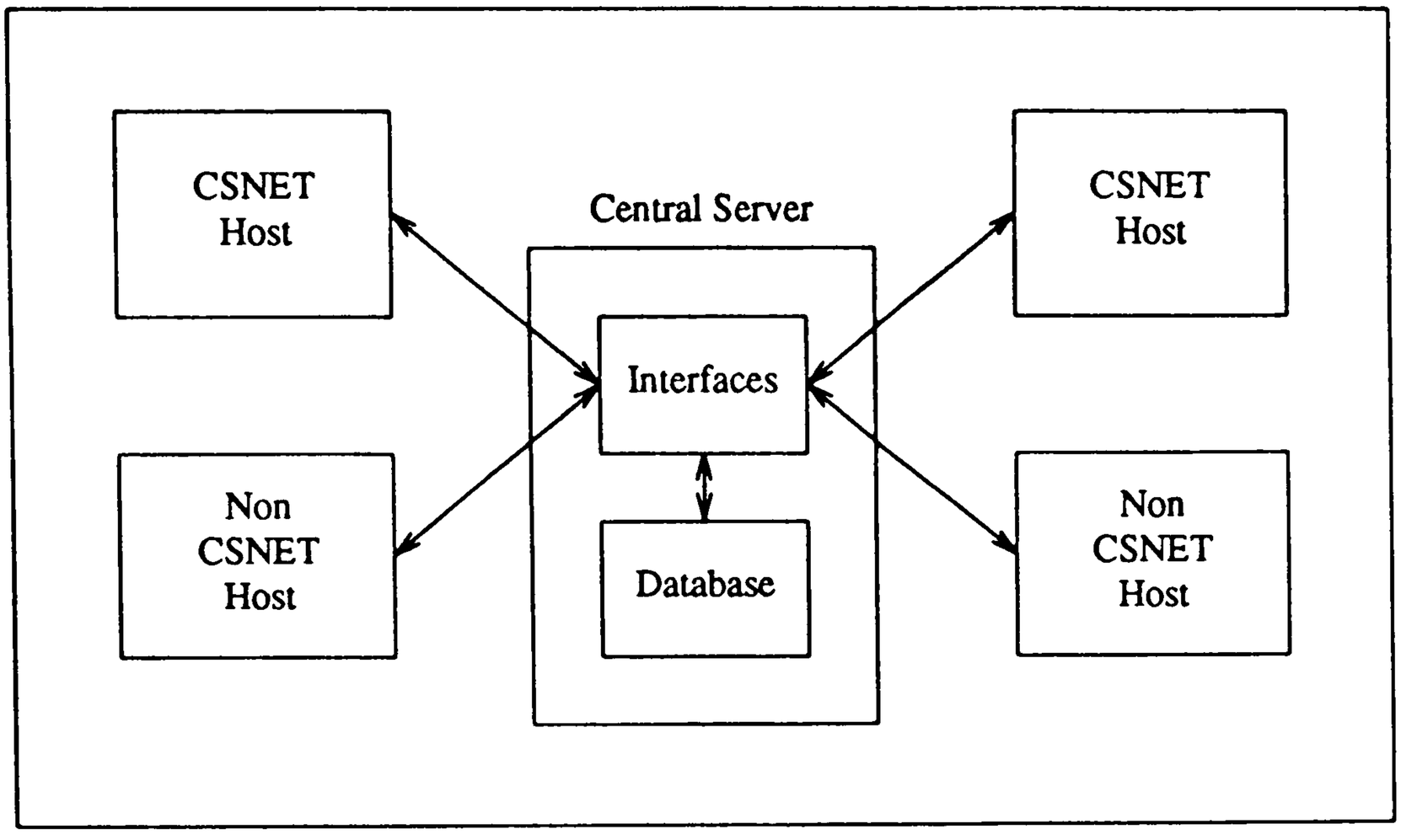


Figure 1.5: The centralised architecture of the CSNET Nameserver.

The following sections describe specific aspects of the CSNET Nameserver in greater detail.

Naming

In order to identify a specific entry within a nameserver query, a user specifies a set of *keywords* matching the information stored within the registry. This set of keywords is divided into *mandatory* and *optional* keywords, interpreted in the following way:

- An entry matches the name if it contains all of the specified mandatory keywords.
- If more than one entry satisfies this criterion, the best match is the one also containing the maximum number of optional keywords.

For example, the following list of keywords might name a registry entry:

mandatory: Benford Nottingham
optional: Computer Science

Keywords describe a flat namespace and sets of keywords may easily be ambiguous. However, the registry unambiguously denotes each entry by a unique identifier. Identifiers are machine oriented and are not intended for human use.

Once a user has identified an entry via a set of keywords, they may bind a *nickname* to the entry for use in future queries. Nicknames are local to each user and are maintained in tables within their agents.

Fields, operations and access control

Each CSNET Nameserver entry is structured as a set of fields from the table below. Fields are specified within the nameserver design and it is not possible for users to define new types of fields.

Nameserver entry fields	
UNIQUE ID	Key uniquely identifying the entry
ACCOUNT	CSNET account name for entry (user.host.site)
MBOX	CSNET electronic mail address of entry owner
CSNPASS	Password for changes to entry from other than home host
FULL NAME	Full name of entry owner
ADDRESS	Post Office address of entry owner
PHONE	Phone numbers of entry owner
MISC	Miscellaneous information about the entry owner

A user may read and manipulate these fields by the following operations, providing general read and update functionality for registry entries.

Major nameserver operations	
WHOIS	Initiate nameserver query
REGISTER	Add a new entry
UNREG	Remove an entry
FETCH	Return a copy of an entry
UPDATE	Change a database entry

All entries within the registry are publically available for reading. However, there is a limited access control mechanism affecting the update of entries. The CSNET Nameserver recognises three classes of entity for update purposes: The *owner* of an entry may register, unregister or update the entry, the *host administrator* may maintain all entries registered for a specific host and the *site administrator* may maintain all entries registered for a site.

1.2.4. Summary of existing nameservers

The previous sections have described three nameservers responsible for the naming and location of resources in a distributed environment. This section identifies the salient features of these systems and concludes by considering the shortcomings rendering them unsuitable for the expanding communications environment of the future.

1.2.4.1. Salient features

The basic function of a nameserver is to map the names of objects or resources onto sets of their properties. The following features are common among existing nameservers.

- Objects of different types (e.g. *users* and *hosts*) are named under the same convention and therefore inhabit the same namespace.
- The namespace is often structured to facilitate the distribution and management of naming. A tree structured namespace is the most common.
- Each object has a unique and unambiguous *distinguished name*. In addition, an object may often have a number of alternative names called *aliases*.
- It is often possible to map from the properties of objects to their names (this is called *generic naming*).
- Several systems allow the on-line update of information and a few support this with an access control mechanism.

Both *the Clearinghouse* and *Bind* are distributed nameservers. The following are notable features of their distributed operation.

- Responsibility for information is divided between a set of servers.
- Servers are arranged in a hierarchical fashion reflecting the namespace.
- Servers exhibit a high degree of autonomy for the maintenance of information.
- Servers may supply clients with hints as to which other servers can perform operations. Servers may also interact directly for administrative purposes.

1.2.4.2. Limitations

The following paragraphs describe areas in which existing nameservers are limited. Many of these limitations, particularly those concerning information management, become critical as the communications environment expands.

Lack of scalability

Those nameservers relying on flat or limited depth hierarchical namespaces are unlikely to cope with the complex global naming environment of the future. There is an argument that a three level hierarchy is sufficient to unambiguously name communication entities on a global scale. However, a successful naming scheme should facilitate the natural management of names. In particular, it should support naming based on organisational structure. A three level hierarchy, like many other fixed schemes, is not flexible enough to describe many such structures and will therefore result in unnatural naming conventions.

Lack of interconnection

Communication between entities from different environments might require the interconnection of different types of nameserver. Currently, there is no interconnection between existing nameservers. Strictly speaking, this is not a criticism of any particular existing system. However, the current situation indicates a clear need for interconnection to achieve a global Directory service. This is the intention of the X.500 standard.

Lack of information management facilitates

Existing systems generally lack mechanisms supporting the management of information. In particular, there is a lack of support for sophisticated access control and data integrity mechanisms.

Lack of sophisticated access control

Many existing nameservers do not support an access control mechanism. Those that do implement a fairly coarse system of control. A sophisticated access control mechanism is required allowing users to dynamically define flexible access controls.

Lack of data integrity control

None of the systems described above support the on-line definition of new classes of property or resource record. Furthermore, there are no mechanisms allowing users to define constraints on the structure and contents of information thus facilitating its consistent update. Without these integrity mechanisms, successful information management on a global scale is virtually impossible.

Lack of system management facilities

Existing nameservers provide little support for nameserver reconfiguration as the communications environment changes. For example, tools are needed to facilitate the introduction and removal of server entities from the distributed system and to manage replicated information. In particular, both BIND and the Clearinghouse specify mechanisms for distributed query navigation. However, there is little support for the management of the *knowledge* information describing the responsibilities of server entities. This information is vital to distributed navigation and requires correct maintenance as the configuration of servers changes.

It is clear from the above that existing nameservers are especially limited in the areas of information and system management. Consequently, this research will consider both the management of communication information and the management of the Directory service itself in detail.

1.3. An introduction to naming

The traditional nameserver role of naming resources in a distributed environment is central to the operation of the Directory service. The Directory provides information about named communication entities and resources. Names provide the handle by which communication information is obtained and communication is possible provided communicating partners know each others names.

Naming has been studied within the general context of filestores and operating systems and a great deal of work exists describing the principles of naming objects within these environments [TER84b, DEM82]. Recent years have seen many attempts to solve the problem of distributed naming, including those forming the basis for the nameservers described in the previous section [WHIT84, SHOC78, SIRB84, SOLL87, MOCK84].

This section identifies those naming issues pertinent to Directory services and outlines a general approach to naming communication entities, based on a global, hierarchical namespace. In particular, it considers support for human oriented naming and the global management of names. However, before discussing different naming mechanisms, it is first necessary to define what we mean by a name.

1.3.1. What is a name?

The following paragraphs give a number of different definitions of the term *name*.

- A *name* is a linguistic object that singles out a particular entity from among a collection of entities [WHIT84].
- A *name* is a [usually human readable] symbol identifying some resource or set of resources [SHOC78].
- A *name* is a linguistic object which singles out a particular object from among all other objects. A name must be unambiguous (i.e. denote just one object) but need not be unique (the only name that unambiguously denotes the object) [CCITT-XDS86].
- A *name* is a binding of a higher level semantic construct to a lower level semantic construct [SALT78].
- A *name* is an object that can be associated with another object and has an equality operation that is reflexive, transitive and symmetric. It has two uses. First, it may provide access to the object with which it has been associated. Second, it may act as a place holder for the object with which it has been associated [SOLL85].

Each of the above definitions describes a particular property of names. It is clear from the first three definitions that the basic function of a name is to identify an object in human terms so that the name can be used to reference the object from among the set of all objects. This property is summarised by saying that a name is *unambiguous*.

The last two definitions describe the idea of a mapping or binding between names and objects and introduce the concept of different *semantic levels* of binding. Shoch illustrates this concept by considering the differences between *routes*, *addresses* and *names*. A route indicates how to get to an object. An address indicates the location of an object in some system but does not tell us how to reach the object. The address may be mapped onto several routes and can be seen as being at a higher (more abstract) semantic level. A name is at a higher level still because it is used to identify the object without describing its location. Names may be mapped onto addresses.

The separation of names from addresses and routes provides the user of a name with an important level of indirection whereby objects can change positions (addresses) transparently from the user point of view. This is known as *location transparency* and is a useful and important property of naming.

This thesis will use the term *name* to refer to a linguistic object, identifying an entity in an unambiguous, but not necessarily unique, manner. In addition, names should exist at a higher semantic level than addresses and routes.

The following sections describe further requirements of names which may be used for the comparison of different naming schemes.

1.3.2. Requirements of naming

Names should meet several requirements for use within computer based communication. Firstly, they should be *user friendly* so that humans can use and remember them. Secondly, they should be suitable for manipulation by computers, mapping them onto sets of objects. Several authors have identified important aspects of human naming including White and Sollins who separately describe the following particularly relevant issues.

- Names must be shareable between communicating partners.
- Names must reflect the naming patterns of different individuals and groups.
- There should be a *multiplicity* of names for an object. This means than an object may have several names valid within different contexts.
- The name of an object should have some human meaning. This means that the name should reflect relevant properties of the object in order to facilitate understanding and memory.

There has been much discussion of the design of naming mechanisms facilitating user friendly naming. It has been argued that names should be designed to make them instantly guessable by human beings. For example, White describes the following properties of user friendliness [WHIT84]:

> "A human should be able to guess an entities name from the information he or she naturally possesses."
>
> "When an entity's name is guessed incorrectly the environment should realise this rather than misinterpreting the guess as the name of another entity."

In addition, it is important to realise the role of *dialogue* between entities (humans or computers) in establishing names for objects. The use of dialogue, gradually establishing a name between a human and the naming system, will be crucial to the realisation of user friendly naming. Dialogue might allow the user to search the name space and allow the system to offer choices and ask questions to resolve ambiguities.

1.3.3. Graphical and hierarchical naming models

This section outlines the design of several distributed naming mechanisms attempting to meet the above requirements.

The objects to be named within a distributed system can be viewed as the vertices of a graph [OPP81]. There are several possible methods of labeling this graph in order to name these vertices.

One could assign a unique label to each vertex thus generating a *flat* or unstructured namespace. In a flat namespace, the name of each object is the same whatever the user's viewpoint. This is known as *absolute* naming.

Alternatively, one could assign a label to each arc joining two vertices. The name of an object, relative to another object, can be constructed by concatenating the labels on the sequence of arcs joining the two vertices. In this scheme, each name starts at an initial node defining a naming *context*. This is known as *relative* naming because the names of objects vary, relative to the initial context chosen. Furthermore, each object may have many different names within each context.

Absolute and relative naming are shown in figure 1.6 below.

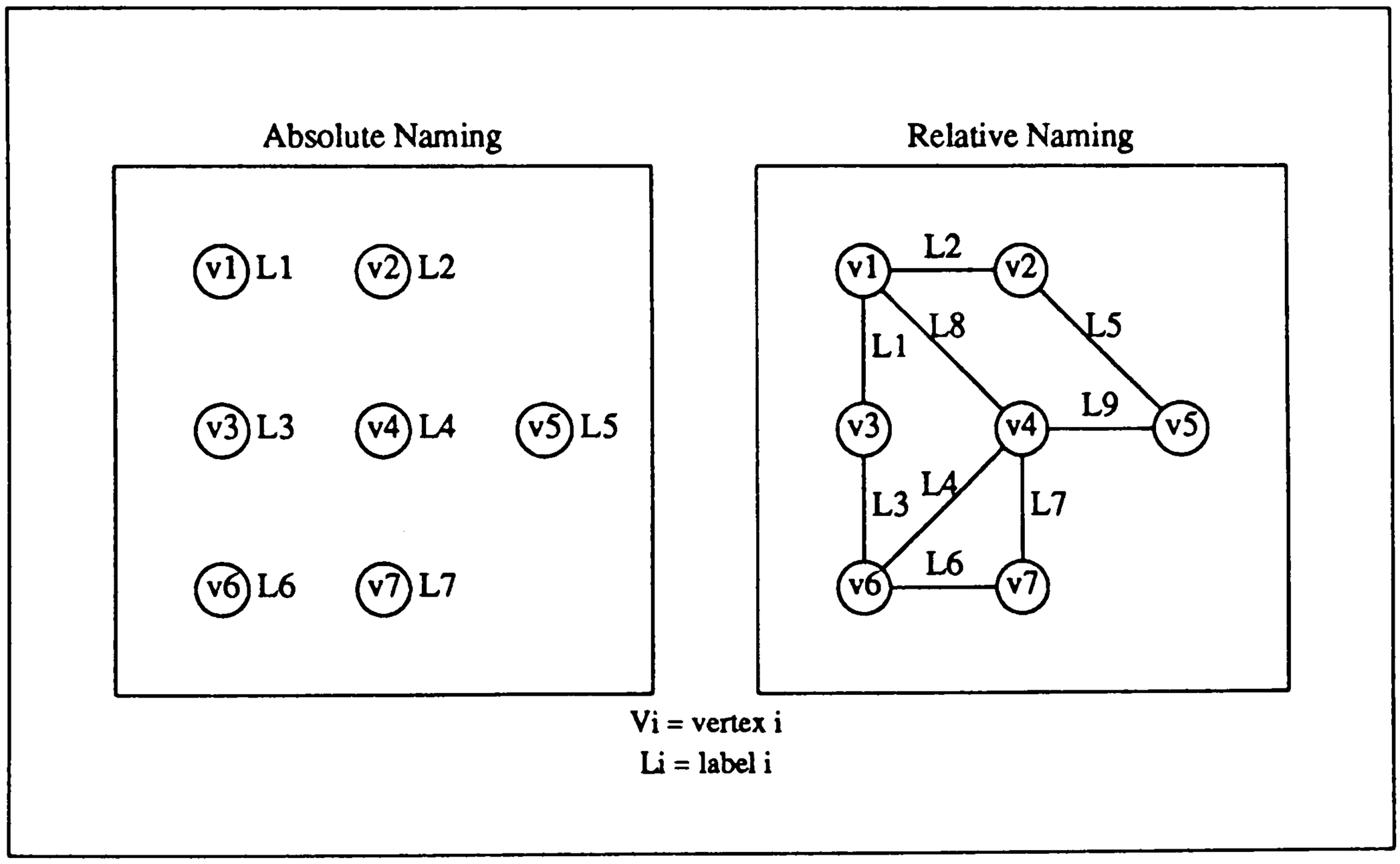


Figure 1.6: Absolute and relative naming

Consider the vertex *v1* in figure 1.6. Under the absolute naming scheme it has the distinguished name *L1* but under the relative naming scheme it may have several names such as *L7, L8* or *L6, L3, L1* from *v7* and *L5, L2* from *v5*.

Absolute naming has the advantage of simplicity. The naming convention is easily understood and provides globally unique names for those applications requiring them. However, a flat namespace becomes difficult to search and manage as the number of named objects increases. It is particularly difficult to assign user friendly names, not conflicting with other names, when the namespace is large.

Relative naming eases the problem of assigning globally unique names in a large namespace at the cost of a more complex model. Applications searching a relative name space may encounter many difficult problems such as *naming loops*.

Both the absolute and relative naming conventions have advantages and disadvantages. In general, neither is obviously superior to the other.

A third possibility, *hierarchical* naming, combines relative and absolute naming to produce a scheme which is simple and also facilitates the mangement of a large namespace. Hierarchical naming schemes represent objects as the vertices of a tree, usually called the *naming tree*. The arcs joining parent vertices to their children are labelled with the *relative names* of the children. Consequently, a globally unique *distinguished name* may be formed by concatenating the sequence of labels on the path from the root of the tree to that node. A typical name tree is shown in figure 1.7.

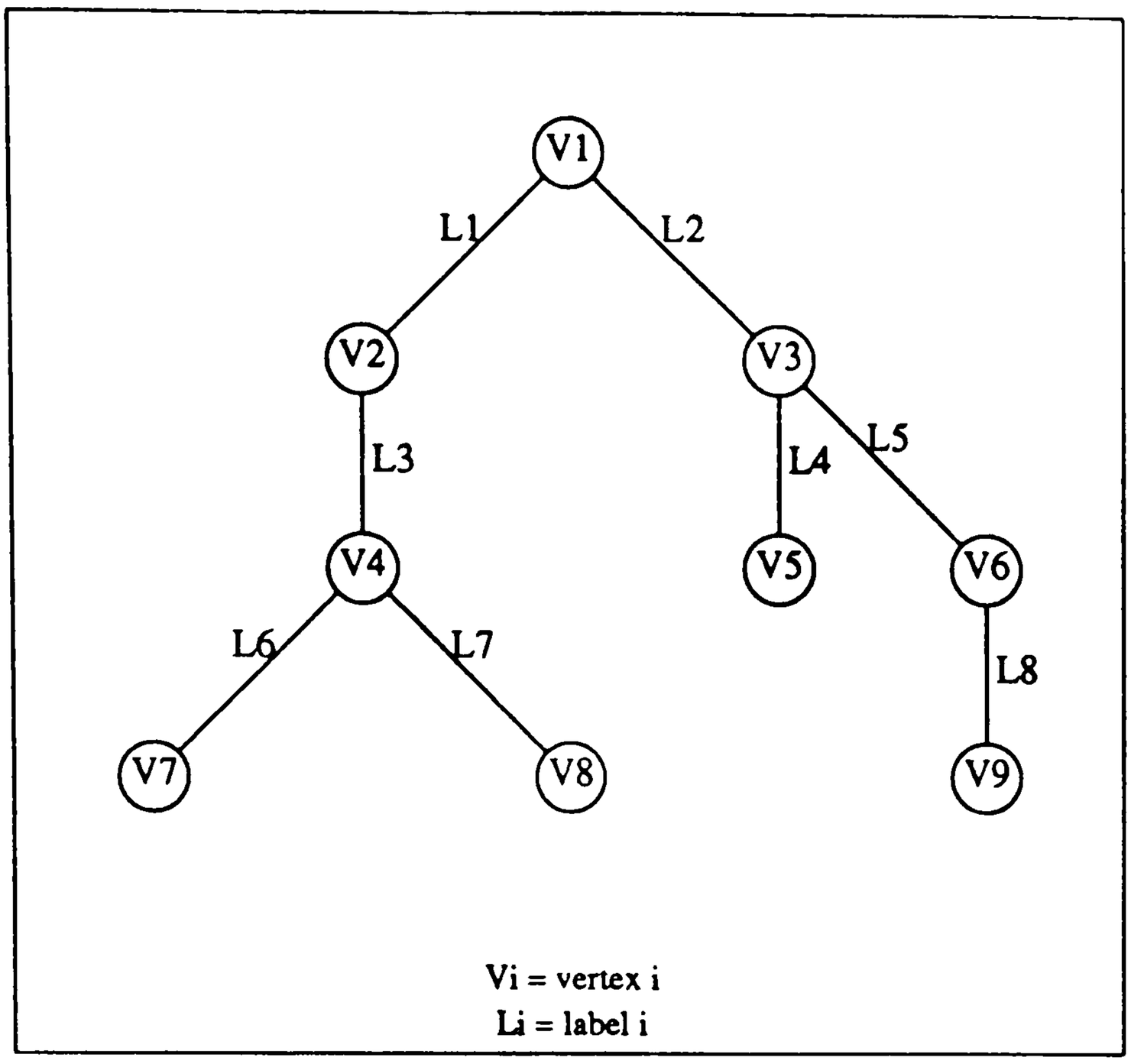


Figure 1.7: A typical name tree

Consider the vertex *v9* in figure 1.7. It has the distinguished name *L2, L5, L8*. Similarly the vertex *v4* has the distinguished name *L1, L3*.

Hierarchical naming has the following advantages:

- Objects have *absolute* distinguished names and can therefore be unambiguously identified.
- The structure of the name space facilitates the management of names and the assignment of new names, even when the name space is very large.
- The naming tree can still support relative naming from some nodes to other nodes (e.g. from parents to children).
- The hierarchy of the naming tree reflects the hierarchical relationships between many objects (e.g. the relationship between a person, their department and the organisation to which they belong).

This last point is perhaps the most important. The key to successful hierarchical naming is its reflection of real world naming conventions. Real world entities generally belong to organisational structures based on a hierarchy. For example, people within a group, groups

within a department, departments within a division and divisions within a company. This point is eloquently made by Pirsig in *Zen and the Art of Motorcycle Maintenance* [PIR74]:

> "... This structure of concepts is formally called a hierarchy and since ancient times has been a structure for all Western knowledge. Kingdoms, empires, churches, armies have all been structured into hierarchies. Modern businesses are so structured."

It should be noted that the hierarchy does not always perfectly fit the real world. For example, a person may belong to two organisations. However, in the majority of cases, it represents a natural and manageable way of naming. Furthermore, limited support for non-hierarchical naming may be achieved by aliasing mechanisms as described later.

In conclusion, hierarchical naming seems the most suitable of the schemes described above for naming communication entities in a globally distributed environment. This is borne out by its use within several existing nameservers (i.e. the *Clearinghouse* and *BIND*) and its inclusion within the X.500 Directory standard. This thesis will also adopt a hierarchical namespace for naming communication entities. This is formally described within chapter 3.

1.4. The ISO/CCITT X.500 standard for Directory services

This section presents an overview of the first joint ISO/CCITT X.500 standard for Directory services, due to be ratified in 1988.

1.4.1. A brief history

The urgent need for a global Directory service has been recognised for some time. As the ISO's *Open Systems Interconnection* (OSI) model for computer networks [ISO-OSI84] is widely adopted in the near future along with various communications standards, such as X.400 for electronic mail [CCITT-X400], this need will become critical.

The general view is that the global Directory will be realised by an interconnection of Directory services provided by a combination of the national PTTs and private organisations. Both the *International Standards Organisation* (ISO) and the *International Telegraph and Telephone Consultative Committee* (CCITT) recognised the need for an international Directory standard to specify this interconnection and initiated standardisation work in the early 1980s. The early work of these committees was strongly influenced by ideas emerging from the *European Computer Manufacturers Association* (ECMA) [ECMA-TR32] and the *International Federation for Information Processing - working group 6.5* (IFIP 6.5) [IFIP-DS83]

who, in turn, were influenced by existing systems such as the *Clearinghouse*. Consequently, early standards output resembled the Clearinghouse model and the output of the ISO and CCITT were similar. As a result, the ISO and CCITT merged their work in 1986 [CCITT-XDS86] and have cooperated since that date to produce a joint standard. This standard is called *X.500*.

Due to the demanding timescale and the complexity of many directory issues, it appears that the 1988 version of X.500 will specify a fairly simple interconnection supporting the retrieval and limited modification of directory information. Issues concerning the management of information and the management of the service itself will be left open for the next study period.

1.4.2. The position of X.500 within OSI

Section 1.1.1 described the future integration of many different services to provide a general communications framework. The Directory service will play a vital role in this integration and it is worth outlining its position within the OSI model and its relationship to other services.

The Directory service resides within the *application layer* (layer 7) of the OSI model and may utilise the supporting OSI stack for its internal operation and interaction with other services. More specifically, the distributed operation of the Directory may utilise the *Remote Operations Service* [ISO-ROS87, CCITT-ROS86] and a number of presentation layer standards such as *ASN.1* [ISO-ASN86]. In addition, the Directory may need to interact with other future application services such as an *authentication service* [MILL87, BRY88].

The interaction of the Directory with other services has been studied by a number of groups, including the ISO, who are producing a general model for *Distributed Office Applications* [ISO-DOA88], and the European Community COST 11-TER funded *AMIGO* project who have produced a distributed group communication model [BOG88]. These models portray the Directory as a supporting service for a number of other services, interacting on a client-server basis.

1.4.3. Overview of X.500

The following pages summarise the major features of the X.500 standard in its present form. It should be noted that, although the standard appears to have achieved a stable state, it may be subject to future revision.

The terms *Directory* or *Directory service* refer to the X.500 Directory service for the remainder of this section.

Scope of X.500

The X.500 standard is described within an *Open Systems* environment allowing the interconnection of systems:

- From different manufacturers,
- Under different managements,
- Of different levels of complexity,
- Of different ages.

The Directory service is intended as an information service for OSI applications and is not a general purpose database system. The Directory has different operational requirements from such systems. For example, the rate of updates to information is expected to be far lower than the rate of retrievals.

For reasons of scale and management, the Directory is a distributed service provided by a number of physically separated application entities called Directory System Agents, each of which knows a part of the total directory information. However, from the user point of view, the logical results of directory operations are usually independent of their location.

The 1988 X.500 Directory service supports the following basic functionality:

- *Read* functionality. This includes the name to attributes mapping.
- *Search* functionality. This involves an attributes to set of names mapping.
- *Modify* functionality. This allows the limited update of information.

The X.500 standard is divided into several sections describing its *information model*, *protocols* and *distributed operations* as well several other issues. These are outlined in the following sections.

Information model

The directory *information model* [CCITT-X501] specifies the abstract structure of directory information. The Directory stores information about *communication entities* which are the humans, groups and application entities taking part in communication. Each communication entity is represented by an *entry* containing the information known about the entity. The set of all entries defines the total information stored by the Directory and is called the *Directory Information Base* (DIB).

Each entry consists of a set of *attributes* representing specific known facts about the entity. For example, an attribute might represent a mail address, a member of a distribution list or a textual description. Each attribute has an *attribute type*, indicating the type of information represented, and a *value*, containing the information. An entry may contain more than one

attribute of a given type. Attribute types are globally unique, being represented by ASN.1 object identifiers.* A number of standard attribute types are defined in [CCITT-X520].

Entries are grouped into *object classes* specifying generic groupings based on the type of communication entity they represent (e.g. *organisational person* or *group of names*). Each entry contains a special attribute of type *object class* indicating to which object class the entry belongs. A number of standard object classes are defined in [CCITT-X521].

Figure 1.8 shows an example entry representing the organisational person *Steve Benford*.

Attribute type	Attribute value
common name	Steve Benford
surname	Benford
object class	organisational person
telephone number	+44 602 506101 x3595
user password	bananas
title	research student

Figure 1.8: Example directory entry

The general structure of entries is shown as part of figure 1.9 below.

Naming

The Directory service employs a hierarchical naming scheme for entries. Entries are arranged into a tree structure reflecting the organisational relationships between the communication entities they represent. This tree structure is called the *Directory Information Tree* (DIT) and is responsible for determining the Directory naming policy.

Each vertex of the DIT is an entry, labeled with a *relative distinguished name* unambiguously identifying it among its siblings. The relative distinguished name (RDN) is composed of a subset of the entry's attributes called *distinguished attributes*. An entry's RDN is assigned by its naming authority, represented by its parent entry in the DIT. Thus, the responsibility for managing names is distributed throughout the DIT.

Each entry has a globally unique and unambiguous *distinguished name*, composed of the ordered sequence of RDNs encountered on the path from the root of the DIT to the entry. Distinguished names provide the basic handle on entries and their contents.

---

*Character strings are usually used in documentation for reasons of legibility.

The relationship between the DIT and entries is shown in figure 1.9 and the relationship between the DIT, relative distinguished names and distinguished names is shown in figure 1.10.

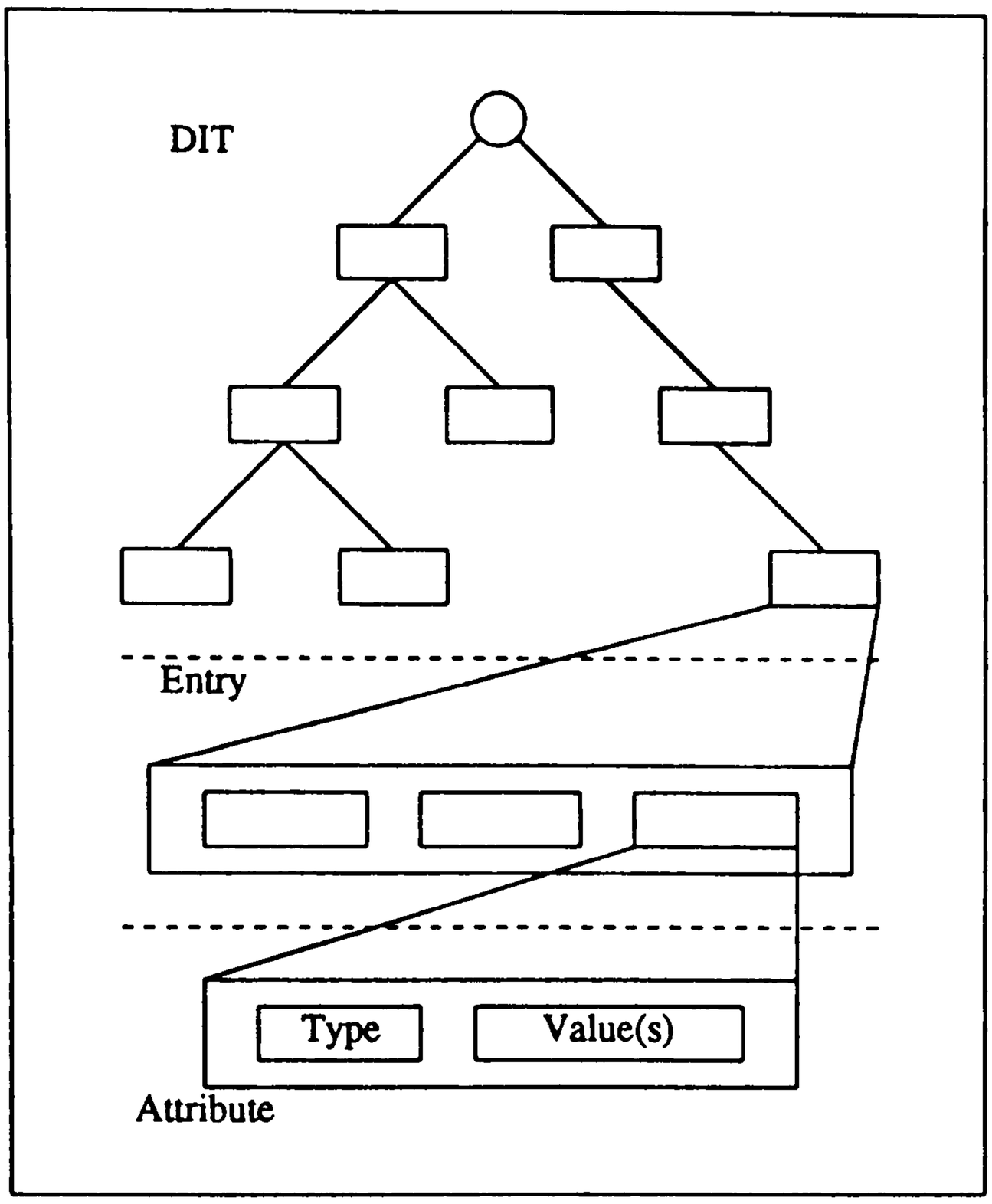


Figure 1.9: General structure of the DIT and entries

Relative distinguished names, and hence distinguished names, are generally chosen to be stable over long periods of time and also to be user friendly where possible. This requires the use of human guessable distinguished attributes.

A distinguished name need not be the only name for an entry. An alternative name, or *alias*, may be supported by the use of special pointer entries called *alias entries*. Alias entries do not contain any attributes other then their relative distinguished names and may only be leaf entries in the DIT.

A directory user names an entry by supplying an ordered set of purported attributes. These are mapped into the desired entry by the process of *name verification*, performing a distributed tree-walk through the DIT. Name verification provides the basic directory name to attribute mapping and indicates whether a name is valid or erroneous. A single name

verification may dereference several aliases during its tree walk. Dereferencing replaces the attributes of the purported name matching an alias with those forming the name of the aliased entry.

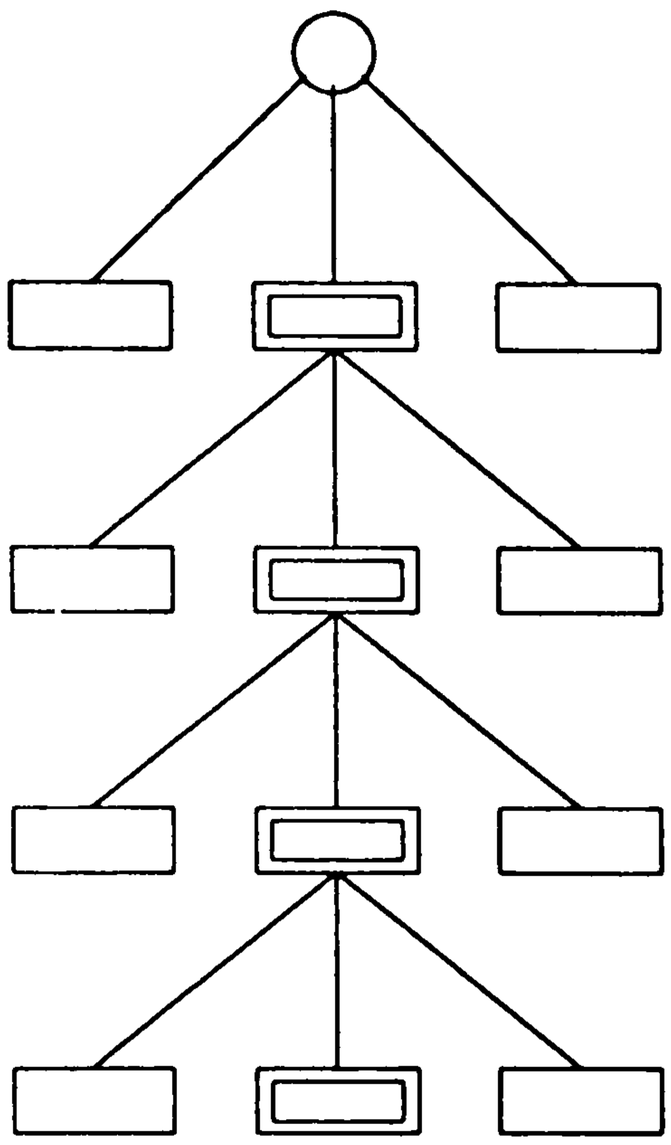
DIT	RDN	Distinguished Name
	{}	{}
	C=GB	C=GB
	O=BT	C=GB,O=BT
	OU=Sales L=Ipswich	C=GB,O=BT, OU=Sales, L=Ipswich
	CN=Smith	C=GB,O=BT OU=Sales, L=Ipswich, CN=Smith

Figure 1.10: The relationship between the DIT, RDNs and DNs

The Abstract Service Definition

The directory *Abstract Service Definition* [CCITT-X511] specifies the user functionality of the Directory service in terms of a set of abstract ports and operations. The operations form the *Directory Access Protocol* (DAP) and provide the user with the functionality to retrieve, search and modify information. The ports and operations defined by the Abstract Service Definition are listed below and briefly described by the following paragraphs.

Abstract ports and operations	
port	operation
Read	Read Compare Abandon
Search	List Search
Modify	Modify Entry Add Entry Remove Entry Modify RDN

General read functionality is achieved via the *Read* port, supporting three operations.

- The *Read* operation returns the values of specified attributes from a single named entry.
- The *Compare* operation returns an indication of whether a named entry contains a specified attribute type/value pair.
- The *Abandon* operation allows the termination of those operations interrogating the Directory.

General browsing of directory information is achieved via the *Search* port, supporting two operations.

- The *List* operation returns the names of the children of a named entry.
- The *Search* operation supports the searching of DIT subtrees for entries matching specific patterns of attributes. The user names a subtree of the DIT, specifies some target attribute types and formulates an expression combining a number of attributes using the logical *and*, *or* and *not* operators. This expression is called a *filter*. The operation returns the values of the target attributes from those entries in the named subtree, matching the filter.

The limited modification of information is achieved via the *Modify* port.

- The *Modify Entry* operation adds, replaces or removes a number of attributes within a named entry.
- The *Add Entry* operation creates a new leaf entry within the DIT.
- The *Remove Entry* operation deletes a leaf entry from the DIT.
- The *Modify Relative Distinguished Name* operation alters the RDN of a named leaf entry.

It is important to note that the latter three operations only apply to entries which will remain as DIT leaves. They do not provide a general facility for building and manipulating the DIT.

Schemas

The structure of the Directory Information Base is governed by a set of rules called *schemas*. These are integrity constraints ensuring that directory information conforms to well defined formats. Schemas specify rules for the following:

- The structure of names and hence the DIT.
- The contents of entries in terms of the attributes they contain.
- Permissible attribute types.
- The syntaxes of attribute values and rules for comparing them.

Each attribute in the Directory is governed by a rule assigning it a unique Object Identifier and specifying its syntax. In addition, this rule states the mechanism by which attributes of this type are compared with one another.

Each entry in the DIT belongs to an *object class*, governed by a schema. This schema specifies *mandatory* and *optional* attributes for entries of this class. Schemas may be nested, allowing more complex object classes to be constructed from a few basic ones.

Naming rules govern which object classes may be children of which others in the DIT and therefore determine possible name forms.

Standard attribute types and object classes are defined in [CCITT-X520] and [CCITT-X521] respectively and any directory operation attempting to violate these rules will fail. The relationship between schemas and the directory information framework is shown in figure 1.11.

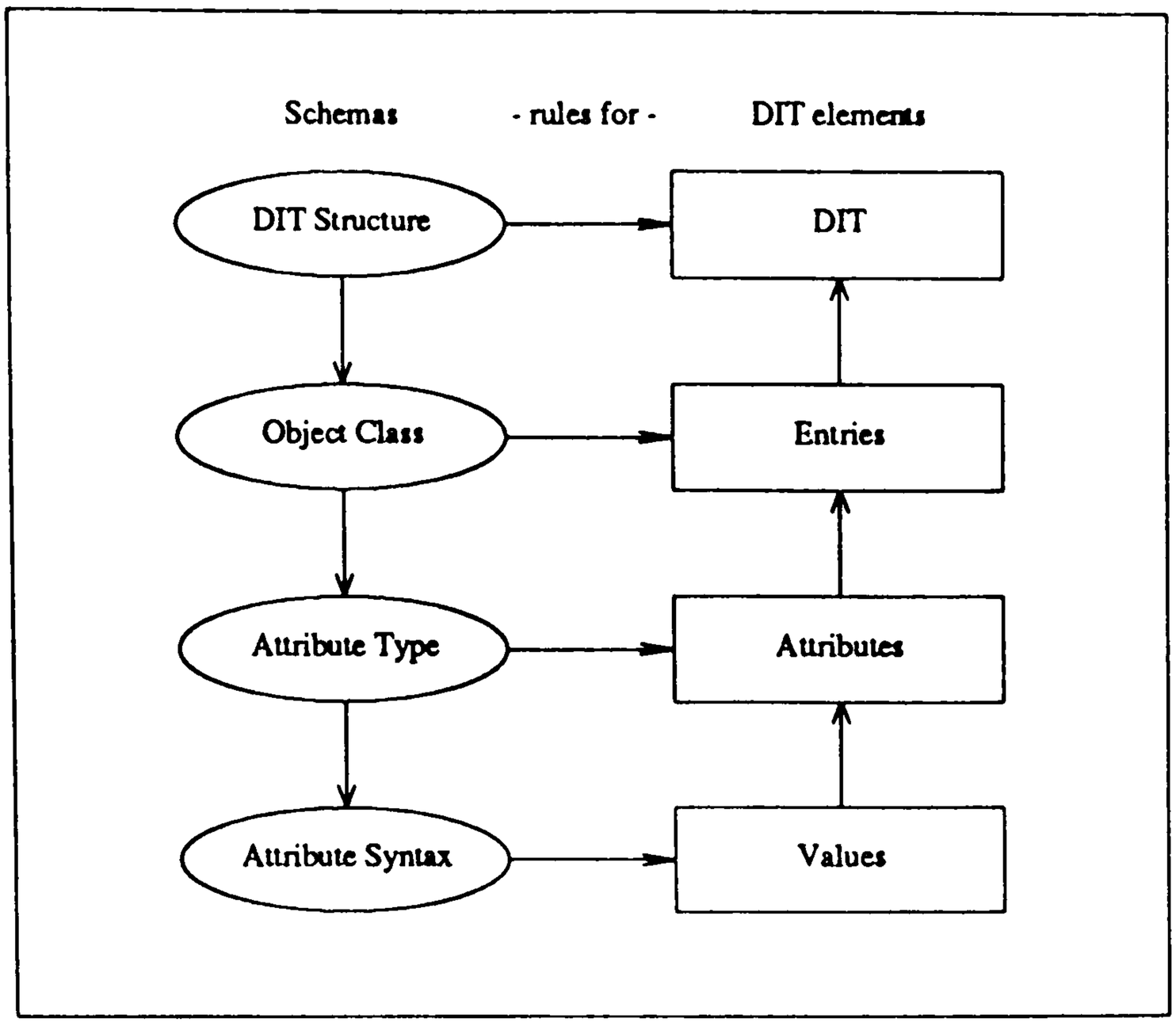


Figure 1.11: The relationship between schemas and the DIT

Functional model and distributed operation

As mentioned above, the global Directory will be a distributed service. The procedures for its distributed operation are specified in [CCITT-X518].

The information constituting the Directory Information Base will be shared between a number of application entities called *Directory System Agents* (DSAs). These cooperate to perform operations, with each DSA knowing a fraction of the total directory information. DSAs can be viewed as a combination of local database functionality and remote interface to users and other DSAs. DSAs may cooperate in order to execute operations. Cooperation may take several forms and requires the navigation of operations through the distributed system. The set of all DSAs forms the *Directory system*.

A user accesses the Directory via an application entity called a *Directory User Agent* (DUA). DUAs manage associations with DSAs and present various interfaces to directory users (human or application). The provision of the Directory service by DUA and DSA functional entities is shown in figure 1.12.

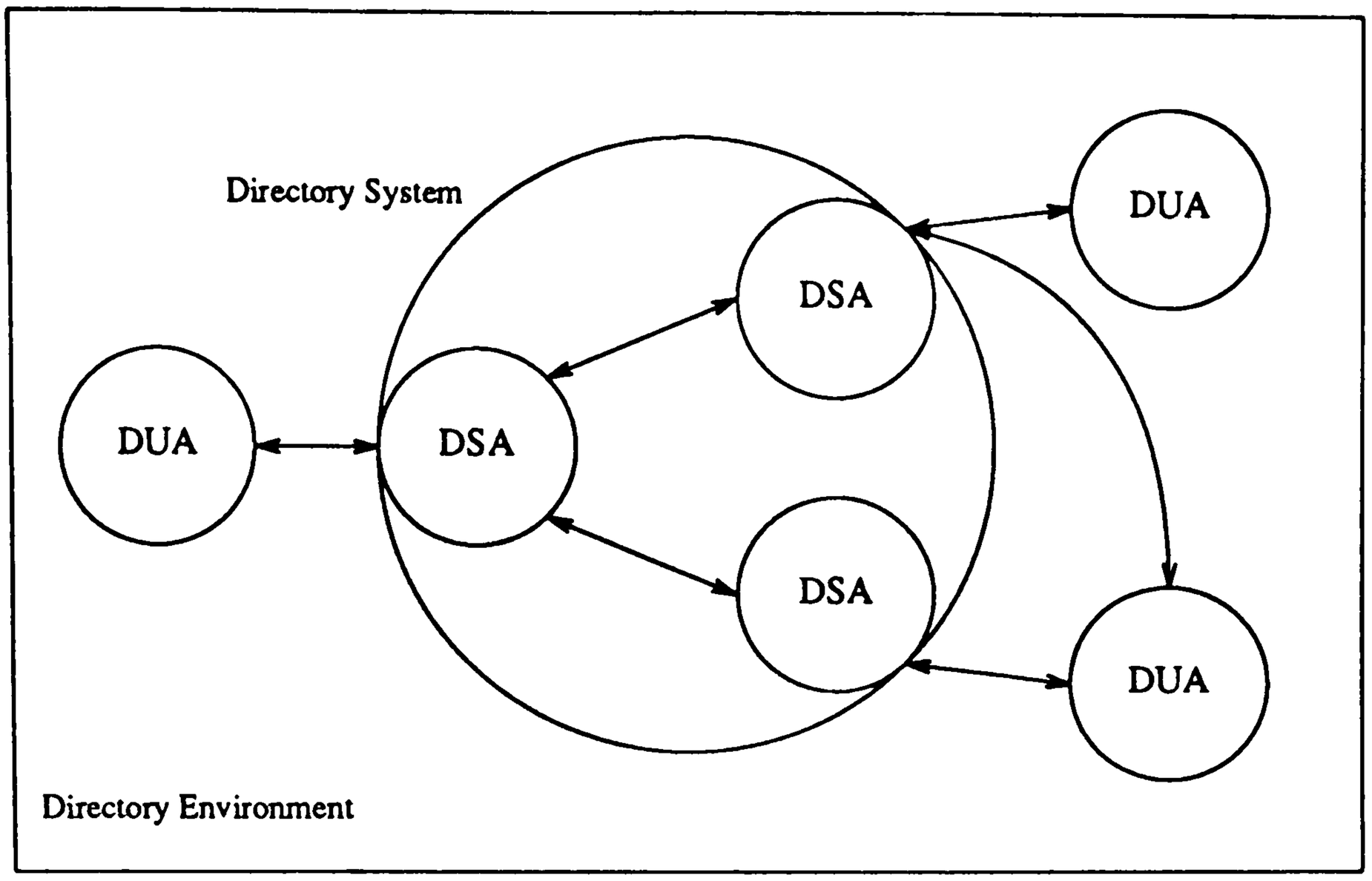


Figure 1.12: The Directory provided by cooperating DUAs and DSAs

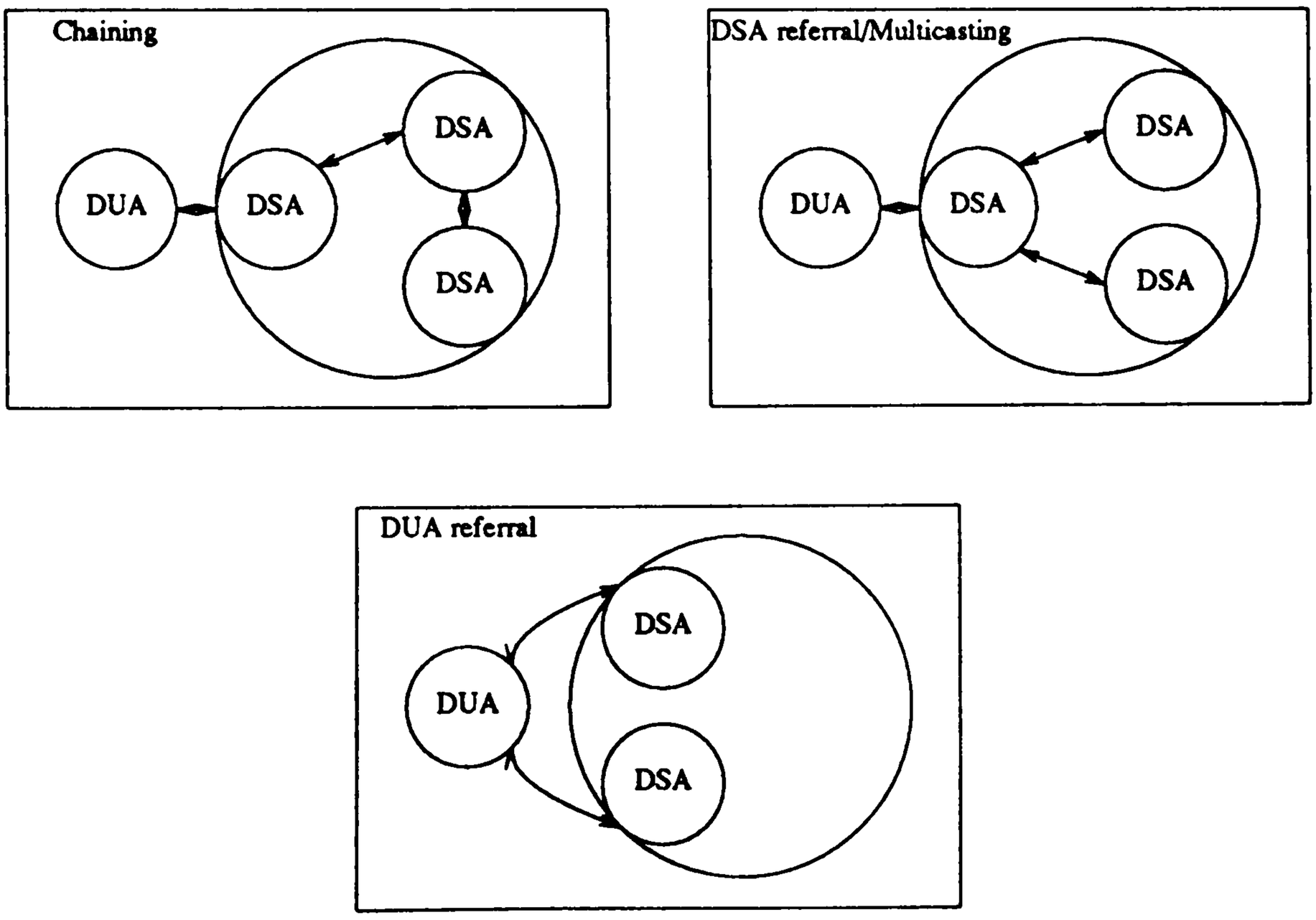


Figure 1.13: Different modes of DSA interaction

A user requests operations via their DUA. These operations are navigated through the Directory system until a set of DSAs is found which can perform them and return the results to the user.

DSAs may utilise several modes of interaction. These are *chaining*, *referrals (DUA or DSA based)* and *multicasting* as shown by figure 1.13

Chaining occurs when DSAs recursively pass the operation to other DSAs which may be able to satisfy the request. The process of locating these *responsible* DSAs is called *navigation*. Once a satisfactory DSA is found, the operation is performed and the results are returned along the chain.

DUA referral involves a DUA contacting a sequence of DSAs to navigate the operation. The result of each contact is either the results and errors associated with the operation or the name and location of another DSA to be contacted (a *referral*).

DSA referral is similar to DUA referral except that a single DSA is responsible for handling referrals instead of a DUA. DSA interactions may also involve *multicasting* where a query is decomposed into subqueries which are transmitted to several DSAs simultaneously. The results are then collected and merged before being returned to the user.

The standard allows DSAs to choose among these methods of interaction depending on conditions, their capabilities and policies.

A detailed description of the distributed operation of the Directory can be found in part 4 of the X.500 standard [CCITT-X518].

1.4.4. Issues requiring further study

The Directory service, specified by the 1988 X.500 standard and outlined in the preceding section, offers what is fundamentally a *read only* service with some limited support for the update of information. Many important issues are left unresolved by the 1988 version of X.500, particularly those concerning information and system management. These will become critical as a global Directory service emerges within the next few years.

The following is a non-exhaustive list of the limitations of the 1988 X.500 Directory standard.

- X.500 does not allow the arbitrary creation and removal of non-leaf entries in the DIT. Thus, the general construction and manipulation of the DIT is not possible via the Directory Access Protocol and must be achieved by local methods.
- X.500 does not support a universal access control mechanism. Instead, access control is left as a local matter although it is briefly discussed in an annex to part 2 [CCITT-X501] of the standard. The lack of support for access controls within the Directory

service represents a severe limitation on information management.

- X.500 does not allow the dynamic definition and management of attribute types, naming rules and schemas as part of the Directory Access Protocol. Instead, new schemas are defined via the standardisation process or during the specification of new applications. This lack of support for *dynamic typing* is another severe constraint on information management.
- Although part 4 of the standard specifies a *knowledge* model describing the responsibilities of DSAs, there is no support for the management of knowledge as the configuration of the Directory system alters. Knowledge management is a vital aspect of a globally distributed Directory service.
- X.500 does not support the replication of information between DSAs and the maintenance of replicated information. Instead, replication is left as a local matter. Replication is vital for improving the robustness and efficiency of the Directory service and is likely to require some standardised support.

It should be noted that some of these issues were discussed within earlier versions of the standard but were dropped, presumably due to the pressing timescale. It is therefore likely that they may reappear within the next study period.

1.5. The relationship of this thesis to the X.500 standard

This thesis describes the specification and implementation of a distributed Directory service with particular emphasis on the issues of information and system management. This work has occurred in parallel with the development of the X.500 standard in the years 1985-88. However, this research has a different emphasis to that of X.500. In particular, the following work is often oriented towards supporting non-standardised and human Directory usage in contrast to X.500 which primarily supports other communication standards such as X.400. Furthermore, this thesis considers the Directory in terms of a distributed database. This approach differs from that of X.500 which is more concerned with protocols for communication.

X.500 will form an international standard moulding the shape of future work in this area. As such, it has been sensible for this research to track the developing standard and to adopt its basic ideas where possible, thus making the results of this thesis directly applicable to the X.500 model. This policy is chosen to increase the relevance of this work to future Directory implementations. In particular, this thesis uses the following X.500 concepts:

- The basic directory information model (i.e. the DIT, entries, attributes and names).
- Those abstract operations reading and searching information.
- The functional model of DUAs and cooperating DSAs.

In addition, the access control model of chapter 4 is based on a model proposed within an earlier version of X.500 [CCITT-XDS86] which is extended and completed. This work includes several contributions to the ISO/CCITT Egham meeting made by the author in cooperation with Alfons Warnking from the GMD, West Germany [BENF86a, BENF86b].

Finally, it should be noted that tracking X.500 has meant following a moving target. Consequently, the basic models used by this thesis are not identical to those within X.500 and often use their own terminology to avoid confusion. Having noted its use of X.500 for basic input, the reader is advised to treat this thesis as self-contained. Its conclusions may then be independently applied to the X.500 standard.

This introductory chapter has demonstrated the need for a global Directory service to support computer based communication and has outlined the roles of the Directory as both information provider and information manager. It has also reviewed the development of Directory services, starting with existing nameservers, and culminating with the proposed ISO/CCITT X.500 Directory standard.

The following chapter begins the specification process by proposing a directory architecture providing a framework for the remainder of the thesis.

Chapter 2

A Layered Directory Architecture

In order to systematically attack the task of specifying and implementing a distributed Directory service, a conceptual framework is required upon which later chapters can build. This chapter develops the necessary framework which is then used to structure the remainder of this thesis. Sections 2.1 to 2.3 describe the framework in terms of a layered directory architecture. Section 2.4 describes the structure of this thesis in terms of the layered architecture and outlines the major issues relevant to each of the following chapters.

The Directory service can be viewed as a specialised distributed database described in terms of a layered architecture. The role of the directory architecture is to provide a framework describing general directory principles independently of specific implementations and to support recognised database ideals such as *data transparency*, *location transparency*, *replication transparency* and *nodal autonomy* as described below.

2.1. Classifying the Directory as a distributed database

The following layered directory architecture has been derived from the ANSI/SPARC architecture describing a general framework for centralised database systems [ANSI75] and the PRECI* architecture describing general distributed database systems [DEEN82]. The proposed architecture falls somewhere between the two in complexity and is specifically oriented towards describing the Directory service.

C.J. Date loosely defines a distributed database by the following:

> "A distributed database is a database which is not stored in its entirety at a single physical location, but rather is spread across a network of locations that are geographically dispersed and connected via communication links" [DATE77].

Distributed databases may be classified by a number of characteristics of which the Directory service exhibits the following:

- The Directory is a *multi-level* distributed database, meaning that it represents a logical collection of data from a number of inter-linked databases resident at a number of *nodes* in a computer network. Each node of a multi-level database has an independent

control system, in contrast to a *single-level* database utilising a single, global control system.

- The Directory is a *canonical* distributed database, meaning that it includes a unified global information model (canonical information model) capable of supporting many external user views of directory information. This canonical information model may be mapped into different *internal* models within individual nodal sub-databases.
- The Directory is a *decentralised* distributed database, meaning that each node holds a copy of the control information governing its interaction with other nodes. This is in contrast to *centralised* systems which are controlled by a single node.

The classification of the Directory as multi-level, canonical and decentralised results from the characteristics of the Open Systems environment within which it operates and the many applications it must support.

- The Open Systems environment means that the Directory service is provided by an interconnection of cooperating, autonomous organisational Directories. Control is therefore multi-level and distributed.
- On the other hand, users of the Directory may need to access information from anywhere within the distributed system and different applications will see the same information framework independently of their location. Thus, the Directory supports a canonical information model.

2.2. Overview of the layered directory architecture

The directory architecture is divided into four distinct layers describing the structure of information at different levels of abstraction ranging from the highly abstract user view to the more concrete local storage view. Each layer concerns different issues and is subject to different requirements, with user functionality issues belonging to the upper layers and implementation and distribution issues belonging to the lower layers.

Before discussing the nature of each layer in greater detail, it is necessary to describe some important properties of general distributed databases, also relevant to the directory architecture.

2.2.1. Requirements of the directory architecture

Data independence

Data independence requires that the user view of directory information is immune to changes in the underlying storage structures and access strategies of supporting software. This means that users should have an abstract view of information, independent of specific implementation tools, and that it should be possible to alter storage mechanisms without having to rewrite applications accessing the Directory.

Location transparency

An extension to the concept of data independence is that of *location transparency* requiring that users are unaware of the physical location of information within the distributed system and are thus shielded from changes to network topology. This is particularly important in an Open Systems environment where no one person has control over the configuration of the network and the partitioning of information between nodes.

Replication transparency

The distributed Directory will store information at a number of nodes and, in the interests of speed and robustness, some information may be stored at more than one node. This is referred to as the *replication* of information. *Replication transparency* requires that a user sees information as if there were only one copy within the entire Directory and is not concerned with problems of updating and accessing multiple copies.

The combined effect of data independence, location transparency and replication transparency is that, in general, the Directory should appear to the user as if it were an abstract information system at a single, virtual host.

Nodal autonomy

Nodal autonomy requires that each node of the distributed system is responsible for its own local information and that overall control of the system is distributed between its nodes. Nodal autonomy generally increases the robustness of a distributed system by removing the dependency on a single, centralised control node.

The following sections discuss the purpose of each layer of the directory architecture. Layers are specified in such a way as to meet the above requirements. The overall architecture is shown in figure 2.1.

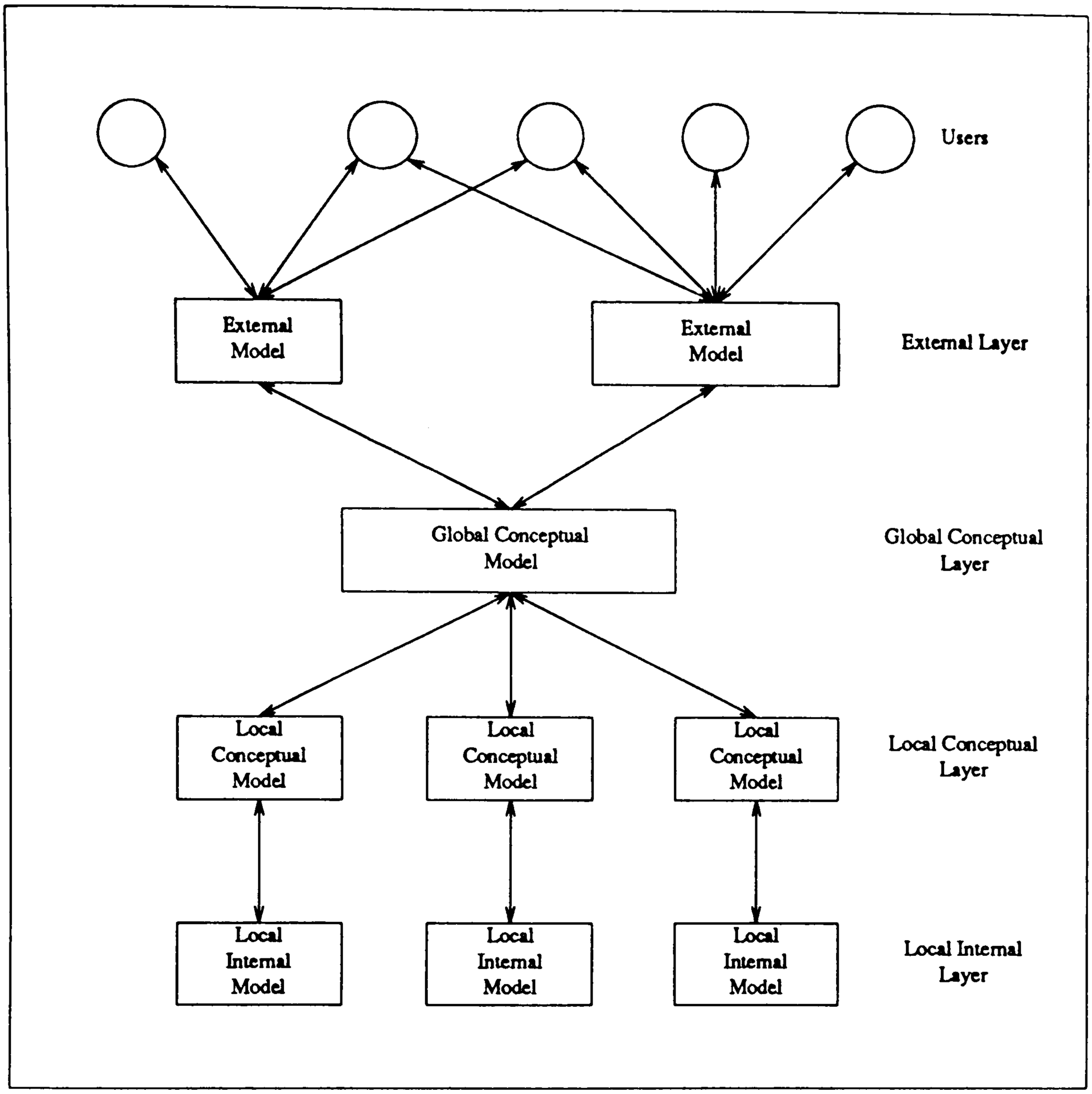


Figure 2.1: The layered architecture of the Directory service

2.2.2. The external layer

The external layer is the one closest to the directory users and concerns the different ways in which specific individuals and applications view directory information. Each user may only be interested in a subset of the total information available and may describe this information using their own specific terms of reference. Furthermore, the way in which users perceive information may be highly abstract when compared to the structure of physical storage.

It is important to note that this thesis applies the term *user* to both the humans and applications accessing the Directory service. For example, a directory user could be the human *Steve Benford* or alternatively, could be an X.400 MTA using the Directory to route a message on behalf of Steve Benford.

In general, a number of users may share a view of the Directory defining the structure of their information and their actions on this information in their own terms. Such a view is called an *external model*. The external layer may contain many different external models corresponding to different types of users. For example, a distribution list application such as the *AMIGO Distribution List* service [BENF87] might define an external model in terms of entities called *distribution lists* and actions such as *add member* and *create list*. On the other hand, a mail application might define an external model in terms such as *mail user* and *name to address mapping*. Furthermore, each individual user may see many external models depending upon the context in which they use the Directory.

The structure of information in each external model is specified by its *external schema* representing information in a manner supporting data independence and location/replication transparency. In general, the term *model* refers to a total view of information and actions whereas the term *schema* only refers to the meta-information defining the structure of information within a particular model.

2.2.3. The global conceptual layer

The global conceptual layer supports a single *global conceptual model*. This is a description of the entire information content of the Directory in an abstract form independent of external models, the distribution of the system and underlying storage and access methods. The global conceptual model is supposed to represent directory information "as it really is". The structure of this information is specified by the *global conceptual schema* supporting data independence and location/replication transparency.

In addition to the global conceptual schema, the global conceptual model includes authorisation checks, access policies and integrity constraints on information. It is these rules and constraints that define each user's view of information and therefore partially specify their external model. Access control mechanisms and integrity constraints are an important part of any large database system and will play a major role in the operation of the global Directory service.

The design of the global conceptual model requires modelling the structure and naming of directory information as well as specifying data access and integrity controls.

2.2.4. The local conceptual layer

The Directory service is a distributed database and, in reality, the information from the global conceptual layer is distributed between a number of distinct database nodes. Each node of the distributed Directory supports a *local conceptual model* describing the information stored and managed by that node as well as information concerning the distribution of the system and the names and responsibilities of other nodes.

Each local conceptual model still views information in the abstract form defined by the global conceptual model. However, only a subset of the entire information is present and the model includes additional *knowledge* information describing the configuration of the system and the partitioning of directory information. This knowledge is used during the navigation of operations to a node or set of nodes able to execute them.

At this level, both location and replication transparency are lost due to the presence of knowledge information. However, the abstract representation of information means that data independence is still preserved.

The local conceptual layer must support nodal autonomy. The modelling of the distributed system should therefore allow decentralised operation.

The design of the local conceptual model requires the partitioning of directory information and the specification of nodal interactions based on local knowledge. Mechanisms for reconfiguring the nodes of the system and managing the subsequent changes to knowledge must also be considered.

2.2.5. The local internal layer

Each node of the distributed Directory must store information and knowledge within some storage system. The choice of local storage systems belongs to local implementors and each node may support its own storage model. For example, relational [CODD70], IMS [DATE77], or CODASYL [CODA71] databases might be chosen as storage systems, each supporting its own view of the stored information. The internal layer is therefore the level at which data independence is finally lost. The storage system's view of directory information and knowledge defines the *local internal model* for each node.

The design of the local internal layer requires the choice of suitable storage systems and the mapping of the local conceptual model to the local internal model at each node.

2.3. Relationship to the functional model

This section explores the relationship between the layered directory architecture, providing a number of models for directory information, and the directory functional model, describing the entities providing the Directory service and the various ways in which they interact.

This thesis assumes the functional model specified by X.500 and reviewed by section 1.4.3. This model defines two classes of entity responsible for implementing the Directory service.

- A *Directory System Agent* (DSA) is responsible for maintaining a local portion of the total directory information and interacting with other DSAs to resolve queries. The set of all DSAs forms the *Directory system* and collectively stores and manages all directory information, called the *Directory Information Base* (DIB).
- A *Directory User Agent* (DUA) provides the interface between the Directory and the user (human or application) and manages the user's association with the Directory system.

A DSA therefore represents a node within the layered architecture and contains a local conceptual model, a local internal model and the mapping between them. The Directory system collectively holds the global conceptual model which is dispersed among its DSAs.

The DUA provides the user interface and therefore presents an external model to the user. It maps this external model into the abstract operations representing actions within the global conceptual model. Figure 2.2 shows the functional model superimposed on the layered architecture indicating the relationship between the information structure, DUAs and DSAs.

The directory functional model defines the *Directory Access Protocol* (DAP) and *Directory System Protocol* (DSP) specifying DUA-DSA and DSA-DSA interactions respectively. In terms of the layered architecture, the DAP describes the abstract actions performed on directory information and therefore defines part of the global conceptual model. The DSP supports the mapping from the global conceptual layer to the local conceptual layer.

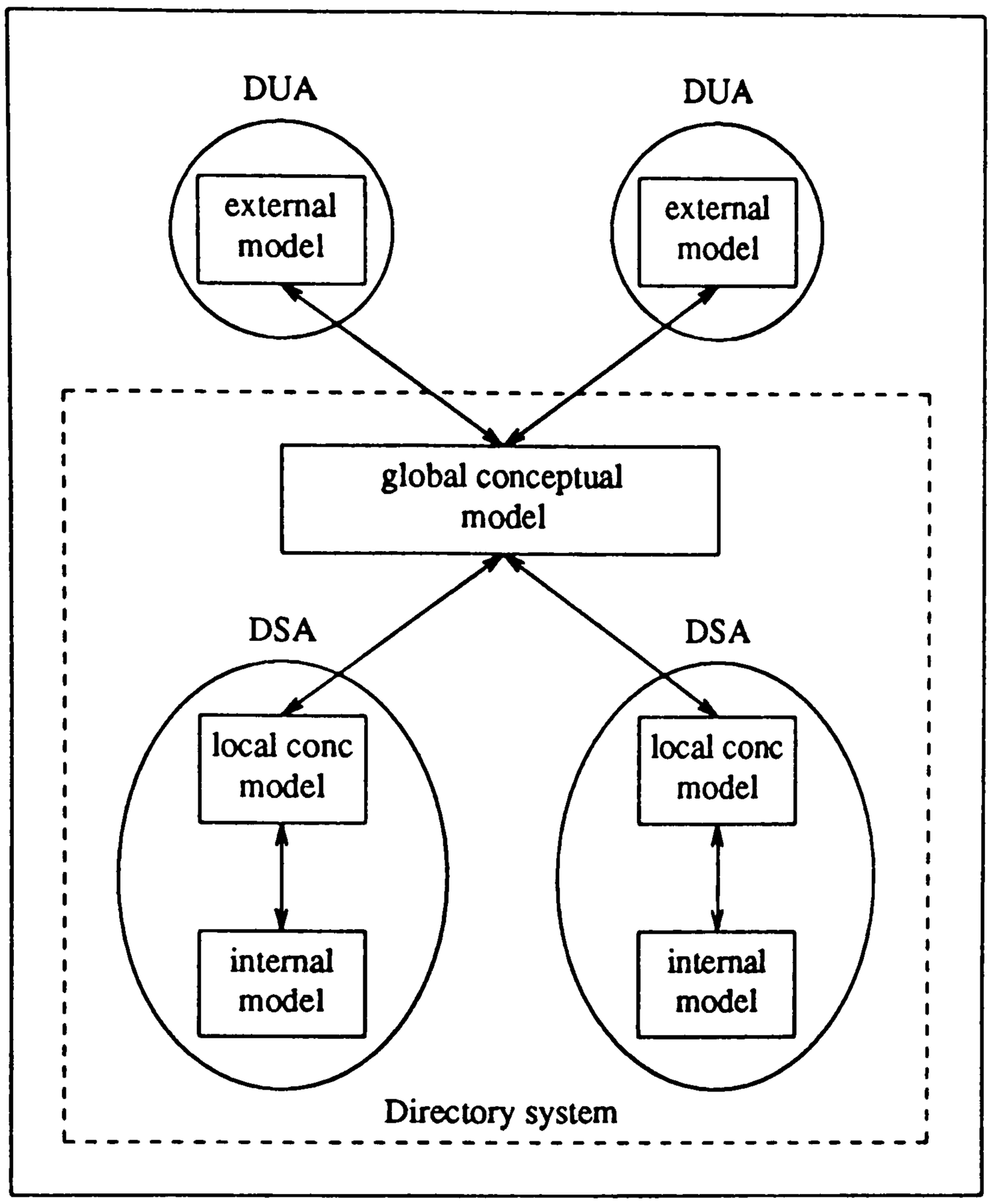


Figure 2.2: Relationship between the directory architecture and functional model

2.4. Using the layered architecture to structure this thesis

The motivation for specifying the layered directory architecture was to provide a framework for structuring this thesis. This section considers the major research issues relevant to each layer of the architecture and explains where they are covered within subsequent chapters.

The overall structure of this thesis moves from the upper to the lower layers and therefore presents a top-down description of the design and implementation of a Directory service. It is worth restating that this does not always follow the chronological progression of my

research where specification and implementation work often occurred in parallel.

The external layer

The external layer concerns the use of the Directory service by specific applications. Research issues relevant to this layer include examining different uses of the Directory, specifying external models and mapping them to the global conceptual model. These issues generally fall outside the scope of this research and there is no chapter dealing solely with the external layer. However, my involvement with the AMIGO MHS+ project has included the specification and implementation of a distribution list protocol including its use of the Directory service [BENF88a]. This is briefly described in chapter 7 when discussing the implementation of a number of Directory User Agents.

The global conceptual layer

The global conceptual layer concerns the design of the directory information model. The following issues are relevant to this work:

- Specifying the abstract structure of directory information.
- Describing the abstract actions applied to directory information.
- Specifying access controls and integrity constraints for directory information.

Chapter 3 covers the first two of these issues. It describes the basic directory information framework in terms of entries, attributes, the Directory Information Tree and a set of abstract operations. This provides the ground work for subsequent chapters.

Chapter 4 examines the issue of information management as applied to the global conceptual model. In particular, it extends the model of chapter 3 to include access controls and integrity constraints supporting the dynamic and flexible management of directory information.

The local conceptual layer

The local conceptual layer concerns the distribution of the global conceptual model between a set of DSAs. The following issues are relevant to this work:

- Partitioning the Directory Information Tree between a set of DSAs and specifying the structure of knowledge.
- The management of knowledge following system reconfiguration.
- The distributed execution of directory operations.

- Supporting replication within the Directory system.

The management issues relevant to the local conceptual layer concern the management of the Directory system itself as opposed to the management of information covered by chapter 4.

Chapter 5 covers the first three of the above issues. It describes the partitioning of the DIT and specifies the structure of knowledge information. It also examines the distributed execution of operations in some detail and describes the distributed support required for the access control and integrity mechanisms. The management of knowledge is also explored within this chapter.

Chapter 6 extends the model of chapter 5 to include the replication of information. It proposes a general directory replication model and examines the support this requires.

The local internal layer

The local internal layer concerns the design and implementation of specific DSAs. This thesis describes the implementation of a DSA based on the RTI Ingres database management system and the ISO Development Environment (ISODE).

The following are the major issues relevant to this work:

- Specifying a DSA internal architecture integrating Ingres and ISODE.
- Mapping the local conceptual schema to the relational data model.
- Implementing distributed navigation and loop control mechanisms.

These issues are covered by chapter 7 of this thesis.

Chapter 3

The Global Conceptual Layer: Information Model and Abstract Operations

This chapter specifies the directory *information model* and *abstract operations* forming the basis of the global conceptual model described previously. The information model fulfils the role of the global conceptual schema by defining the structure of directory information. The set of abstract operations describe the actions performed on directory information.

The specification of the information model and abstract operations requires the following three steps:

- Representing communication entities in terms of directory entries (section 3.1).
- Specifying a global naming scheme for communication entities (section 3.2).
- Describing operations to read, search and manipulate directory information (section 3.3).

Chapter 4 extends this basic model to include access control and integrity mechanisms supporting the management of directory information.

The model described by this chapter closely resembles that of X.500 in terms of the Directory Information Tree and some abstract operations. However, there are differences between the two models. In particular, this thesis specifies several new operations. A thorough grasp of this work is essential to understanding later chapters and, for this reason, the basic directory information model is fully described below. Furthermore, this chapter defines its own names for operations in order to avoid confusion with those of X.500 which, although similar, exhibit a number of important differences.

3.1. Communication entities as entries and attributes

The role of the Directory service is to store and manage information about *communication entities* which are the humans, applications and devices taking part in communication. Each communication entity is viewed as a distinct real world object and is represented in the Directory by a single, named *entry*. Examples of communication entities are *Steve Benford* (person), *Beth* (laser printer) and *Amigo Group* (distribution list).

Each communication entity may exhibit a number of properties, represented by a set of *attributes* within its entry. An attribute has a type, identifying a generic class of property, and a value, containing a specific instance of that property. An entry may contain more than one attribute of the same type except where constraints specifically limit an attribute type to having a single value. For example, an entry representing a person might contain attributes of type *common name* and *mail address* and an entry representing a distribution list might contain attributes of type *member* and *moderator*.

Attribute types and values are generally depicted by character strings throughout this work for reasons of legibility. Their exact structure is described in chapter 4.

In its most unstructured form, the directory information model can therefore be viewed as sets of entries containing sets of attributes each of which has a type and value. The set of all directory entries is called the *Directory Information Base* (DIB) and figure 3.1 shows an example part of a DIB.

Entries may be divided into generic groupings called *object classes* where each object class represents a distinct type of real world communication entity. Examples of object classes might be *person, device, application entity* or *distribution list*. The object class of an entry is represented by an attribute of type *class*.

entry for person "Steve"

class	person
office	1302
phone	3595
mail	sdb@cs.nott
title	research student

entry for list "Social"

auditor	sdb@cs.nott
member	jpo@cs.nott
member	hugh@cs.nott
member	dave@cs.nott
member	sdb@cs.nott
class	list

entry for entity "Nott Mta"

class	MTA
mail	cs.nott.ac.uk
owner	admin
desc.	X.400 MTA

entry for device "Beth"

class	printer
name	Beth
location	room 1101
owner	admin
serial no	12547896

entry for organisation "Nott"

label	Nott. Uni
class	org
phone	484848
location	Nottingham

entry for person "Hugh"

class	person
office	1301
phone	3647
phone	3595
mail	hugh@cs.nott
title	lecturer

Figure 3.1: Example Directory Information Base

3.2. Naming communication entities

The DIB outlined above loosely resembles a relational style data model. However, in order to support the global management of names, a hierarchical naming structure is superimposed on this basic entry and attribute model. The reasons for adopting a hierarchical naming scheme were discussed in section 1.3 and the following points briefly summarise the approach of this thesis to the naming problem.

Human naming is a complex process and is reliant on a number of factors such as *context* and *dialogue*. This research does not address the problem of supporting general human oriented naming in distributed systems but considers the following more limited goals constituting a step in this direction.

1 Allow the naming of communication entities such that the responsibility for managing names is distributed in a natural way.

2 Support globally unambiguous names which may be manipulated by computer systems.

3 Support the separation of names from addresses and routes.

4 Provide a single, unified naming scheme freeing users from accessing many different, idiosyncratic application naming schemes.

5 Support limited dialogue and searching facilities so that users may resolve names by a combination of information they possess and hints from the system.

The following section begins by specifying an abstract, hierarchical naming scheme satisfying goals 1-4 and deals with goal 5 via a combination of browsing and searching facilities described later.

The Directory Information Tree

Each directory entry represents a unique real world object and should therefore have at least one unambiguous name by which humans and the system refer to it. It is likely that the responsibility for naming will be distributed along the lines of organisational hierarchy and this is reflected in the directory naming scheme.

The entries of the Directory Information Base can be arranged into a tree structure called the *Directory Information Tree* (DIT) where each vertex of the tree represents a communication entity or a naming authority. Examples of naming authorities are *countries*, *organisations* and *organisational units* and these typically form non-leaf vertices of the DIT responsible for allocating the names of their immediate children. This separation of naming authorities ensures that the responsibility for naming is distributed throughout the global Directory service thus making possible the allocation and management of globally unambiguous names.

Each entry in the DIT has a *relative distinguished name* (RDN) consisting of a specially marked set of its attributes called *naming attributes*. These are chosen by the parent naming authority so that the entry's RDN is unambiguous among all of its siblings, thus allowing the children of a naming authority to be distinguished from each other.

A globally unique and unambiguous name may be constructed for each entry by forming the ordered sequence of RDNs on the path from the root of the DIT to the named entry. This sequence is called the entry's *distinguished name* (DN). Each entry has exactly one distinguished name.

Figure 3.2 shows an example Directory Information Tree indicating naming authorities, RDNs and DNs. It also indicates that the naming attributes for each child of a naming authority need not have the same attribute types (e.g. the children of */C=GB /O= Nott.Uni/).*

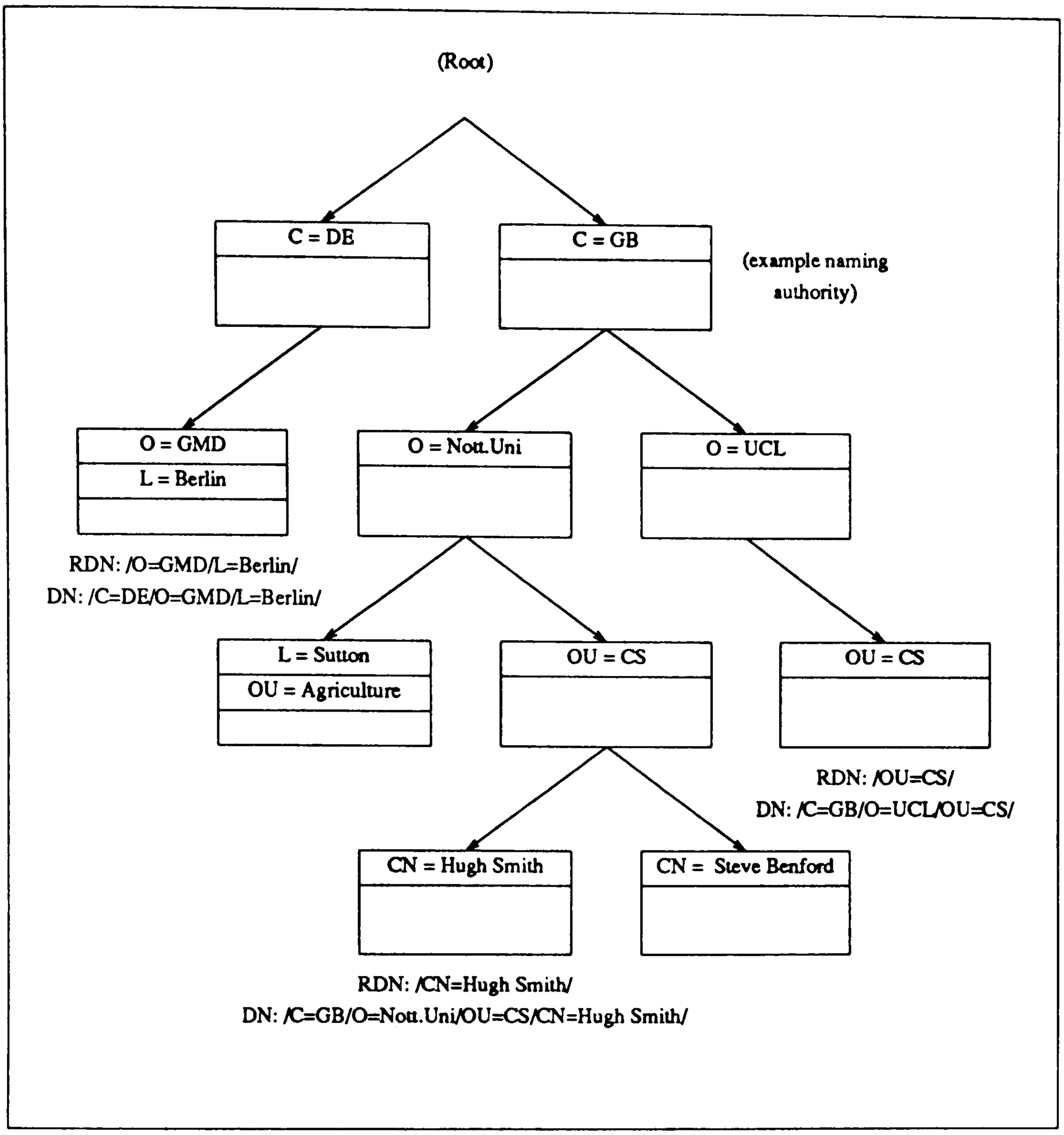


Figure 3.2: Example Directory Information Tree

The remainder of this thesis adopts the following notation for representing directory names within text. Distinguished names and relative distinguished names are ordered sequences of attributes and can be represented by ordered character strings with "/" symbols separating attributes shown as *type = value* sub-strings. For example, the distinguished name of the entry representing *Steve Benford* in figure 3.2 is */C=GB /O=Nott.Uni /OU=CS /CN=Steve Benford/*.

This notation is summarised by the following B.N.F description:

<coded name> ::= <attribute sequence> "/"
<attribute sequence> ::= <attribute> <attribute sequence> | <attribute>
<attribute> ::= "/" <type> "=" <value>
<type> ::= sequence of any ASCII characters
<value> ::= sequence of any ASCII characters

The above diagram and the remainder of this chapter use the following abbreviations for attribute types:

Key to attribute types	
C	Country
O	Organisation
OU	Organisational Unit
L	Location
CN	Common Name

The Directory service defines a one-to-one mapping between the set of distinguished names and the set of directory entries. The process of mapping from a name to the named entry is called *name verification* and is perhaps the most fundamental action of the Directory, occurring at least once during every abstract operation. Name verification takes a *purported name*, specified by a user as an ordered set of possible attributes, and performs a distributed tree walk, starting at the root of the DIT, in an attempt to locate an entry matching the name. This process may either identify the purported name as being valid or erroneous.

Aliases

In addition to its distinguished name an entry may be identified by a number of alternative names called *aliases*. An alias is the distinguished name of a special pointer entry in the DIT containing the name of a referenced entry called the *aliased object*. An alias may point at any DIT entry and an alias entry is empty except for the attributes forming its relative distinguished name.

The name verification procedure may *dereference* a number of aliases during a tree walk to resolve a purported name. For each alias encountered, the procedure jumps to the referenced aliased object entry and continues verifying the purported name as before. Abstract operations manipulating aliases may turn off dereferencing during name verification in order to access the alias entry itself instead of the aliased object entry.

The proposed naming scheme does not inherently support *alias back-pointers*. This means that the aliased object entry need not necessarily contain the names of its aliases. The justification for this decision is that once a user has gained access to a name it is their choice

as to how they use it and specifically, whether they set aliases for that name. Consequently, the use of aliases may be a personal matter and they need not be visible from the aliased object entry. Furthermore, the maintenance of back-pointers might impose a large overhead on directory implementations.

It should be noted that aliases may point to other aliases. This might provide a useful level of indirection during the naming process. For example, a local alias might point to the chairman of a comittee. This, in turn, could be an alias pointing at the current person occupying the position of chairman. In this case, the local alias would automatically map to the current chairman at any given time.

On the other hand, the aliasing of aliases introduces the problem of *alias loops*. For example, two aliases might point at each other resulting in an infinite loop during name verification. However, unlike X.500, this thesis does not forbid the aliasing of aliases.

- Firstly, disallowing aliases for aliases is an unnecessarily restrictive convention, removing potentially useful functionality.
- Secondly, it does not solve the problem of alias loops as they can still occur by other means (see section 3.4).

Instead, the problem of alias loops is better addressed by mechanisms within the Directory system. This is discussed within chapter 5

Figure 3.3 shows aliases in the Directory Information Tree.

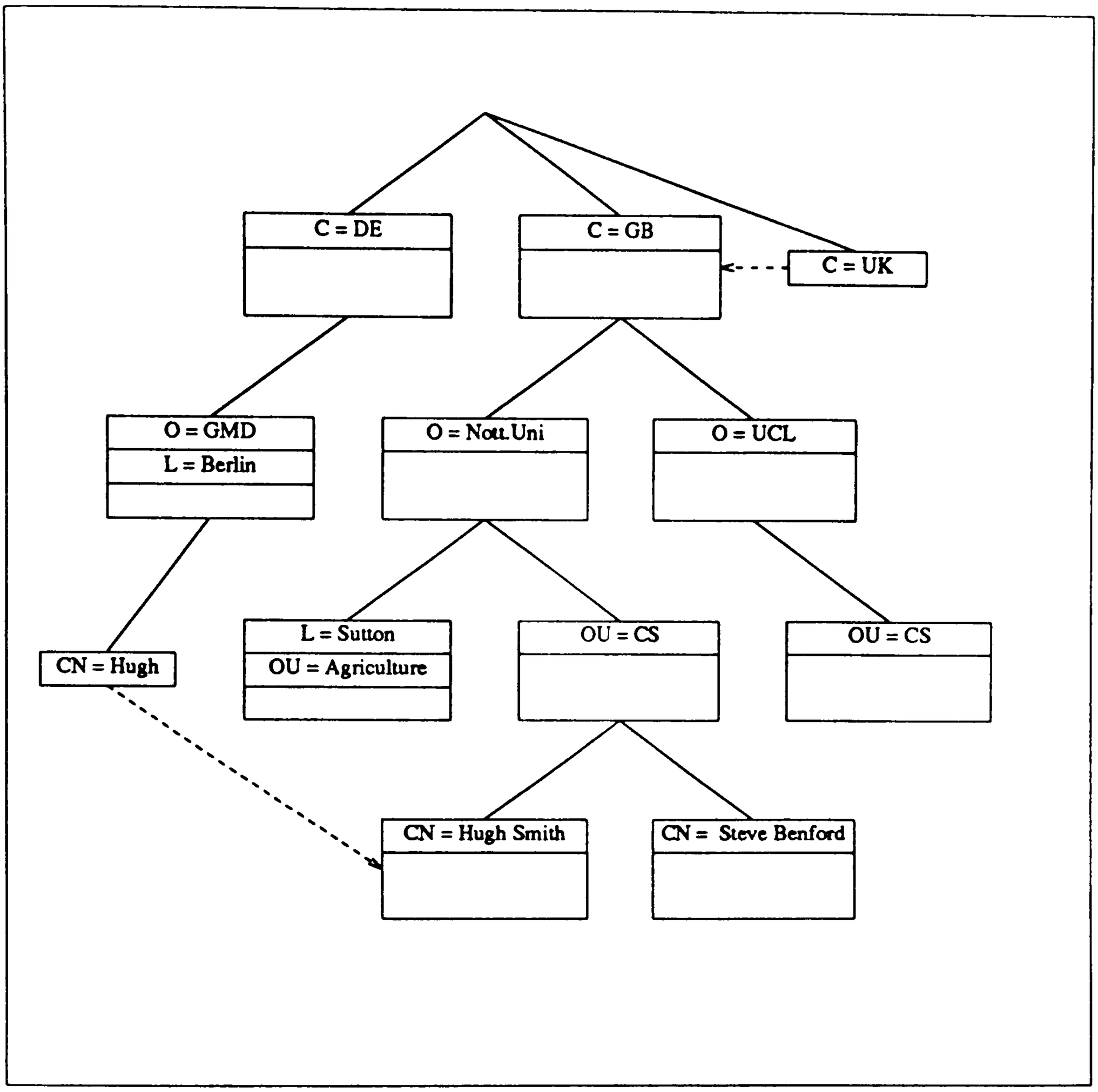


Figure 3.3: Aliases in the Directory Information Tree

If we consider figure 3.3 the purported name */C=DE /O=GMD /L=Berlin /CN=Hugh/* would be mapped to the entry having the distinguished name */C=GB /O=Nott.Uni /OU=CS/ CN=Hugh Smith/*. Another example maps the purported name */C=UK /O=Nott.Uni /OU=CS/* to the entry having the distinguished name */C=GB /O=Nott.Uni /OU=CS/*.

The uniqueness of distinguished names allows the Directory to determine whether any two aliases describe the same entry by resolving them and syntactically matching the resulting DNs.

3.3. Abstract operations

The previous sections have described the abstract structure of directory information in terms of the Directory Information Tree, entries and attributes. This section describes the actions which read, search and manipulate these structures by specifying a number of *abstract operations*. Abstract operations form the interface to the global conceptual model and hence define the protocol between the DUA and DSA elements of the Directory as shown in figure 2.2. This protocol is called the *Directory Access Protocol* (DAP) and is supported by the *Remote Operations Service* (ROS) [ISO-ROS87, CCITT-ROS86], a Remote Procedure Call like mechanism allowing a client application to request operations at a server application during an *association* between the two. Each of these abstract operations may have an argument set supplied by the client and return either a result set or error set depending upon the actions of the server.

This work and the remainder of this thesis specify the arguments, results and errors of abstract operations in terms of the *Abstract Syntax Notation One* (ASN.1) [ISO-ASN86]. This notation is extensively used throughout this thesis to describe general structures and operations.

The following sections describe the abstract operations listed in the table below and indicate the steps taken by the Directory to perform each operation as well as the ASN.1 structures involved.

Basic Abstract Operations
Read Entry*
List Subordinates*
Search*
Modify Entry*
Add Entry*
Delete Entry*
Add Alias
Delete Alias
Suspend Entry
Reinstate Entry
Reset Alias
Bind*
Unbind*

*These operations have similar counterparts in X.500.

3.3.1. Reading directory information

The Directory service provides the *white pages* name to attributes mapping via the *Read Entry* operation. This takes a purported name and a list of target attribute types and returns the values of the requested attributes from the named entry. The Directory must first verify the purported name and then read the attributes from the located entry. These are returned to the user along with the distinguished name of the entry. Errors might arise from an erroneous name or from the specification of non-existent attribute types and it is possible for the Read Entry operation to return a combination of results and errors (e.g. when some of the requested attributes are present and others are missing).

The following ASN.1 definition outlines the arguments, results and errors relevant to the Read Entry operation. This operation will be enhanced by later chapters and a full specification of all structures is given in Appendix A.

```
ReadEntry ::= ABSTRACT-OPERATION
        ARGUMENT ReadArgument
        RESULT ReadResult
        ERRORS { NameError, AttributeError }

ReadArgument ::= SET {
        entry [0] Name,
        types [1] SET OF AttributeType }

ReadResult ::= SET {
        [0] EntryInformation OPTIONAL,
        [1] SET OF ReadError }

EntryInformation ::= SET {
        name [0] DistinguishedName,
        attributes [1] SET OF AttributeInformation }

AttributeInformation ::= SET {
        type [0] AttributeType,
        values [1] SET OF AttributeValue }
```

3.3.2. Browsing directory information

Users may wish to browse the Directory, exploring the structure of the Directory Information Tree and searching portions of the tree for entries matching specific patterns of attributes. These facilities are provided by a combination of the *List Subordinates* and *Search* abstract operations respectively.

List Subordinates takes the name of an entry and returns the names of its immediate subordinate entries (i.e. its children) and can therefore be used for stepwise exploration of the DIT.

```
ListSubordinates ::= ABSTRACT-OPERATION
        ARGUMENT ListArgument
        RESULT ListResult
        ERRORS { NameError }

ListArgument ::= Name

ListResult ::= SET OF DistinguishedName
```

The Search operation provides the *yellow pages* attributes to names mapping. It allows the user to supply a search expression based on combinations of attributes, determines the names of the entries matching the expression and reads a list of target attributes from these entries. The search expression or *filter* is a set of patterns of the form *type = value* specifying attributes contained within matching entries. These patterns may be recursively combined by the logical *AND*, *OR* and *NOT* operators giving a powerful selection mechanism. Filters are also discussed in section 4.2 of this thesis.

The search operation also allows the user to constrain the scope of the search to a specific portion of the DIT by naming a subtree where searching will occur. Further constraints may be applied limiting the search to a specified depth within this subtree.

```
Search ::= ABSTRACT-OPERATION
        ARGUMENT SearchArgument
        RESULT SearchResult
        ERRORS { NameError, AttributeError }

SearchArgument ::= SET {
        subtree [0] Name,
        level [1] CHOICE {
                fixed [0] INTEGER,
                unlimited [1] NULL DEFAULT },
        filter [2] Filter OPTIONAL,
        targets [3] SET OF AttributeType }

Filter ::= CHOICE {
        item [0] SEQUENCE {
                AttributeType,
                AttributeValue },
        and [1] SET OF Filter,
        or [2] SET OF Filter,
        not [3] Filter}

SearchResult ::= SET {
        entries [0] SET OF EntryInformation,
        errors [1] SET OF ReadError}
```

As an example use of the Search operation, consider searching for the entries of all "professors" at "The University of Nottingham". This would require a fully recursive search of the subtree with root */C=GB/ O=Nott.Uni/* matching the expression *class = person AND title =*

professor. This Search operation can be viewed as an extended Read Entry operation, reading from sets of entries as opposed to individual entries.

3.3.3. Modifying the contents of entries

It will be necessary to modify the attributes within existing entries to reflect changes in the real world communication entities they represent. These modifications may require the addition of new attributes to an entry, the replacement of old attribute values with new attribute values or the removal of attributes from an entry. All of these functions are subsumed by a single *Modify Entry* operation allowing users to perform several modifications to a single, named entry. Modify Entry is supplied with the name of the entry to be modified and a number of modifications specifying new attributes, replacement attributes and attributes to be deleted. The Directory locates the named entry and implements the modifications subject to the following constraints:

- Attributes forming the relative distinguished name of the entry may not be modified.
- Attributes constrained to be single valued must remain so.

This operation either returns the NULL result indicating success or a number of possible errors indicating failure.

```
ModifyEntry ::= ABSTRACT-OPERATION
        ARGUMENT ModifyArgument
        RESULT ModifyResult
        ERRORS { NameError, AttributeError, ModifyError }

ModifyArgument ::= SET {
        entry [0] Name,
        modifications [1] SET OF Modification }

Modification ::= SEQUENCE {
        type AttributeType,
        CHOICE {
                add [0] AttributeValue,
                delete [1] AttributeValue,
                replace [2] SEQUENCE {
                        oldvalue AttributeValue,
                        newvalue AttributeValue} } }

        ModifyResult ::= NULL
```

The Modify Entry operation groups several logical modifications into a single operation for two main reasons:

- Firstly, name verification occurs only once thus improving efficiency.

- Secondly, the user might wish to view the modifications as a single logical unit or *atomic transaction*.

The latter suggests the need for a directory "transaction control policy" [KOH81] allowing logical modifications to be grouped into indivisible transactions. In general, the Directory must react consistently when a single operation results in a mixture of errors and successful updates. For example, what should be the result of attempting to add two values for an attribute constrained to be single valued?

This thesis proposes that each abstract operation is viewed as an atomic transaction and that the Directory assumes a *one out - all out* transaction control policy whereby if any single modification fails then all modifications within the operation fail. This solution has the advantage of simplicity and consistency and is assumed for all operations modifying the Directory Information Base in any way. The need for higher level, user specified transaction control is discussed in section 3.5 below.

3.3.4. Adding and deleting entries

The *Add Entry* and *Delete Entry* operations are responsible for creating and removing directory entries.

Add Entry creates a leaf node in the DIT complete with an initial set of attributes. The operation is supplied with the name of the parent entry, the proposed RDN for the new entry and the initial attributes for the entry. The new entry is created subject to the following conditions:

- The proposed RDN for the new entry must be unique among its siblings.
- Attributes constrained to be single valued must remain so.

It is important to note that the new entry may represent a naming authority and hence need not always remain as a leaf of the DIT. Thus, the Add Entry operation allows general DIT construction.

Transaction control applies to the entire operation on a one out - all out basis.

```
AddEntry ::= ABSTRACT-OPERATION
        ARGUMENT AddEntryArgument
        RESULT AddEntryResult
        ERRORS { NameError, RDNError, AttributeError }

AddEntryArgument ::= SET {
        parent [0] Name,
        rdn [1] SET OF Attribute,
        initial-info [2] SET OF Attribute }

AddEntryResult ::= NULL
```

The Delete Entry operation removes a leaf entry from the DIT. This operation is supplied with the name of the entry to be deleted and the Directory is responsible for deleting all attributes associated with the entry. The problem of removing or resetting aliases associated with the entry to be deleted is addressed by section 3.4.

```
DeleteEntry ::= ABSTRACT-OPERATION
        ARGUMENT DeleteEntryArgument
        RESULT DeleteEntryResult
        ERRORS { NameError }

DeleteEntryArgument ::= Name

DeleteEntryResult ::= NULL
```

3.3.5. Adding and deleting aliases

Aliases may be created and removed via the *Add Alias* and *Delete Alias* operations.

Add Alias is supplied with the name of the parent of the alias entry, the RDN of the new alias entry and the name of the aliased object entry. An alias entry is created and associated with the distinguished name of the aliased object entry subject to the following conditions:

- The proposed RDN for the alias entry must be unique among its siblings.
- The name of the aliased object must designate a valid entry in the DIT.

The Delete Alias operation deletes a named alias entry from the DIT. This requires that dereferencing does not occur during verification of the supplied alias name so that the Directory may manipulate the alias entry and not the aliased object entry.

```
AddAlias ::= ABSTRACT-OPERATION
        ARGUMENT AddAliasArgument
        RESULT AddAliasResult
        ERRORS { NameError, RDNError, AliasError }

AddAliasArgument ::= SET {
        parent [0] Name,
        rdn [1] SET OF Attribute,
        aliased-object [2] Name }

AddAliasResult ::= NULL

DeleteAlias ::= ABSTRACT-OPERATION
        ARGUMENT DeleteAliasArgument
        RESULT DeleteAliasResult
        ERRORS { NameError }

DeleteAliasArgument ::= Name

DeleteAliasResult ::= NULL
```

3.4. Suspending directory information

The structure of the Directory Information Tree reflects the hierarchical structure of organisations and users are typically named on an organisational basis. Although relative distinguished names should be chosen to be stable over long periods of time it is clear that they will change as people migrate between jobs and organisations. Many human relationships will survive migration and communicating partners will need to be informed of name changes so that they are still able to communicate. One symptom of this problem is that aliases may need to be reset following name changes. However, the Add Entry and Delete Entry operations do not support the functionality to do this. In particular, the deletion of an entry and its subsequent replacement by an alias of the same name might result in an alias loop. This demonstrates that alias loops might occur, following changes to the DIT, even if aliases may not initially point at other aliases.

Present day communications services such as paper mail often advertise changes of name and address by extra-directory mechanisms such as the distribution of special messages to close associates. However, these mechanisms depend on the efforts of the individual and tend to be unreliable. It is also traditional to leave a *forwarding address* at a previous location and this has been introduced into the telephone service where recorded messages may inform callers of a change in number for a subsequent period of time. There is a parallel between these lower layer services and a requirement for the Directory service to gracefully accommodate name changes.

The Directory service could support the migration of communication entities and their subsequent name changes by allowing the *suspension* of names. Suspension can be viewed as the *phasing out* of information over a period of time. Instead of being deleted from the Directory Information Tree an entry could be marked as suspended and could be associated with a new name. References to the suspended entry would result in a name error, returning the new name of the communication entity to the user who would be free to retry the operation if they wished. If the error resulted from the use of an alias, the user might choose to reset the alias to point at the new name. After a suitable period of time the suspended entry could be permanently deleted, after which future references to it would generate the usual name errors.

It would be possible to build a Directory service to recognise suspended information and automatically reset aliases and restart operations without consulting the user. This would be undesirable for two main reasons:

- A user should be informed of a change of name because it has human meaning, unlike a change of system address of which users should be unaware.
- A user may not wish to continue communication following a change in name and should be free to delete aliases instead of resetting them.

The suspension of entries means that there will be a gradual transition from use of an old name to use of a new name and that users will adapt to name changes in a natural way as they encounter them during communication. In addition, suspension allows aliases to be reset before an entry is deleted thus preventing alias loops from accidently occurring.

It should also be possible to *reinstate* a suspended entry without deleting it and its contents. Only leaf entries of the DIT may be suspended and reinstated.

The suspension and reinstatement of entries and the resetting of aliases can be supported by the following three abstract operations: *Suspend Entry* takes the name of an old entry, the name of a new entry and a deletion date. It marks the entry as suspended and associates it with the new name without deleting its contents. The Directory then deletes the marked entry on the specified date. *Reinstate Entry* returns the named suspended entry to active status. The *Reset Alias* operation takes an alias name and the name of a non-alias entry in the DIT and resets the alias to point at the new entry.

```
SuspendEntry ::= ABSTRACT-OPERATION
        ARGUMENT SuspendEntryArgument
        RESULT SuspendEntryResult
        ERRORS { NameError, SuspendError }

SuspendEntryArgument ::= SET {
        entry [0] Name,
        new-name [1] DistinguishedName,
        date [2] GeneralisedTime }

ReinstateEntry ::= ABSTRACT-OPERATION
        ARGUMENT ReinstateEntryArgument
        RESULT ReinstateEntryResult
        ERRORS { NameError, SuspendError }

ReinstateEntryArgument ::= Name

ResetAlias ::= ABSTRACT-OPERATION
        ARGUMENT ResetAliasArgument
        RESULT ResetAliasResult
        ERRORS { NameError, AliasError }

ResetAliasArgument ::= SET {
        alias [0] Name,
        newobject [1] DistinguishedName }

SuspendEntryResult ::= NULL
```

ReinstateEntryResult ::= NULL
ResetAliasResult ::= NULL

3.5. The role of user specified transaction control

The abstract operations specified above describe the fundamental actions applied to directory information and are subject to the *one out - all out* transaction control policy specified in section 3.3.3. However, actions within a user's external model may appear to be quite different from those within the global conceptual model (i.e. the abstract operations). In particular, it is possible that, as more complex applications are developed, external actions may involve complex sequences of interlinked abstract operations. This idea is familiar from general database systems where users construct arbitrarily complex queries from database query languages such as SQL [IBM81]. In order to allow user actions to appear as indivisible units, these languages include statements supporting user specified transaction controls.

An important issue is whether the Directory Access Protocol should support user specified transaction control instead of the one out - all out policy described above. User specified transaction control would allow the logical grouping of sequences of abstract operations to form higher level functions. However, the implementation of such a mechanism within the highly autonomous distributed environment of the Directory service would be extremely difficult due to problems of maintaining consistency*. It is therefore unlikely that a globally distributed Directory, using current technology, would be able to support such functionality except on a limited scale. Consequently, this thesis proposes that the Directory service should not support user specified transaction controls.

It is important to realise that this decision places a limitation on the ability of the Directory to support arbitrarily complex usage and that applications requiring sophisticated data manipulation tools may need the support of other distributed information services. The designers of distributed applications using the Directory should be aware of its limitations in this area.

This concludes the specification of the fundamental directory global conceptual model. The following chapter discusses the issue of information management as applied to this model.

---

*Consistent distributed updates are discussed in chapter 6.

Chapter 4

The Management of Directory Information

The *management* of communication information has been identified as a major goal of the Directory service (section 1.1.3). This chapter extends the basic global conceptual model of chapter 3 to include access control and integrity mechanisms supporting the flexible management of directory information.

The abstract operations specified in chapter 3 provide the functionality to update directory information. However, without effective information management they will not support the cohesive maintenance of information on a global scale and their unchecked use would soon result in chaos. Management is therefore concerned with the update of information within a *management framework*, defining policies and constraints which ensure that updates are correct and meaningful. The goal of this chapter is to specify the required management framework.

- Section 4.1 provides an overview of information management in terms of data access control and data integrity. It explains how the Directory can support many different management frameworks based on global access control and integrity mechanisms.
- Section 4.2 specifies a data access control mechanism to be included within the global conceptual model. It uses existing access control theory to derive a directory access control mechanism.
- Section 4.3 specifies a directory data integrity mechanism supporting the dynamic specification of rules constraining the structure of directory information.
- Finally, section 4.4 presents a brief summary of the directory global conceptual layer as specified in chapters 3 and 4.

It should be noted that the management of directory information described by this chapter is a different issue to the management of the Directory service itself addressed by chapters 5 and 6.

4.1. Mechanisms supporting the management of directory information

The management of information is a complex topic covering a wide range of issues such as access control, authorisation, security and integrity. Many of these issues have been explored within the context of general databases and operating systems [CCITT-X518, TAN87, DATE77]. This thesis examines two particular aspects of information management as applied to the Directory service. Broadly speaking, it considers information management in terms of the twin issues of *access control* and *data integrity* although it recognises that these are just two facets of the overall management problem.

The terms "access control" and "data integrity" may have different meanings in various contexts. This chapter loosely defines them in the following way:

- Access control concerns the protection of information against "unauthorised disclosure, alteration or destruction" [DATE77]. More specifically, access controls specify administrative policies for information in terms of which users may take which actions on what information.
- Data integrity is concerned with maintaining the correctness of information. Integrity mechanisms attempt to ensure that information is accurate and meaningful at all times according to explicitly specified rules or policies.

In general, access control mechanisms protect against the *illegal* alteration or destruction of information by ensuring that only trusted users may perform updates whereas integrity mechanisms protect against the *invalid* alteration or destruction of information by enforcing rules governing its structure and contents.

The work in this chapter is strongly influenced by two major factors affecting the design of both the access control and integrity mechanisms. These may be summarised by the terms *dynamic management* and *Information Management Domains* and are explained by the following paragraphs.

Dynamic management

The Directory service operates in a dynamic environment of many organisations and applications. This environment will be continually evolving and, consequently, directory information management policies will be constantly changing. Directory management mechanisms must therefore support the *dynamic management* of information by allowing the dynamic definition of new access controls and data integrity constraints. In particular, the process of defining new policies should be explicitly supported by the Directory Access Protocol.

Information Management Domains

Due to the global scale of the Directory service and its provision by many autonomous organisations, a single directory information management framework is not feasible. Instead, the management of directory information can be considered in terms of a number of autonomous *Information Management Domains* (IMDs) each of which supports it own management framework of access control and integrity policies. Information Management Domains reflect logical administrative boundaries, typically corresponding to organisational boundaries. For example, the University of Nottingham might be represented by an IMD defining local security and integrity constraints closely following its specific organisational structure.

IMDs belong to the external layer of the directory architecture and are not explicitly represented within the directory global conceptual layer. However, the management mechanisms belonging to the global conceptual model must be flexible enough to simultaneously support many different external IMDs.

Figure 4.1 shows how operations updating directory information are subject to the management policies of different Information Management Domains which are themselves mapped onto the access control and integrity mechanisms belonging to the global conceptual model.

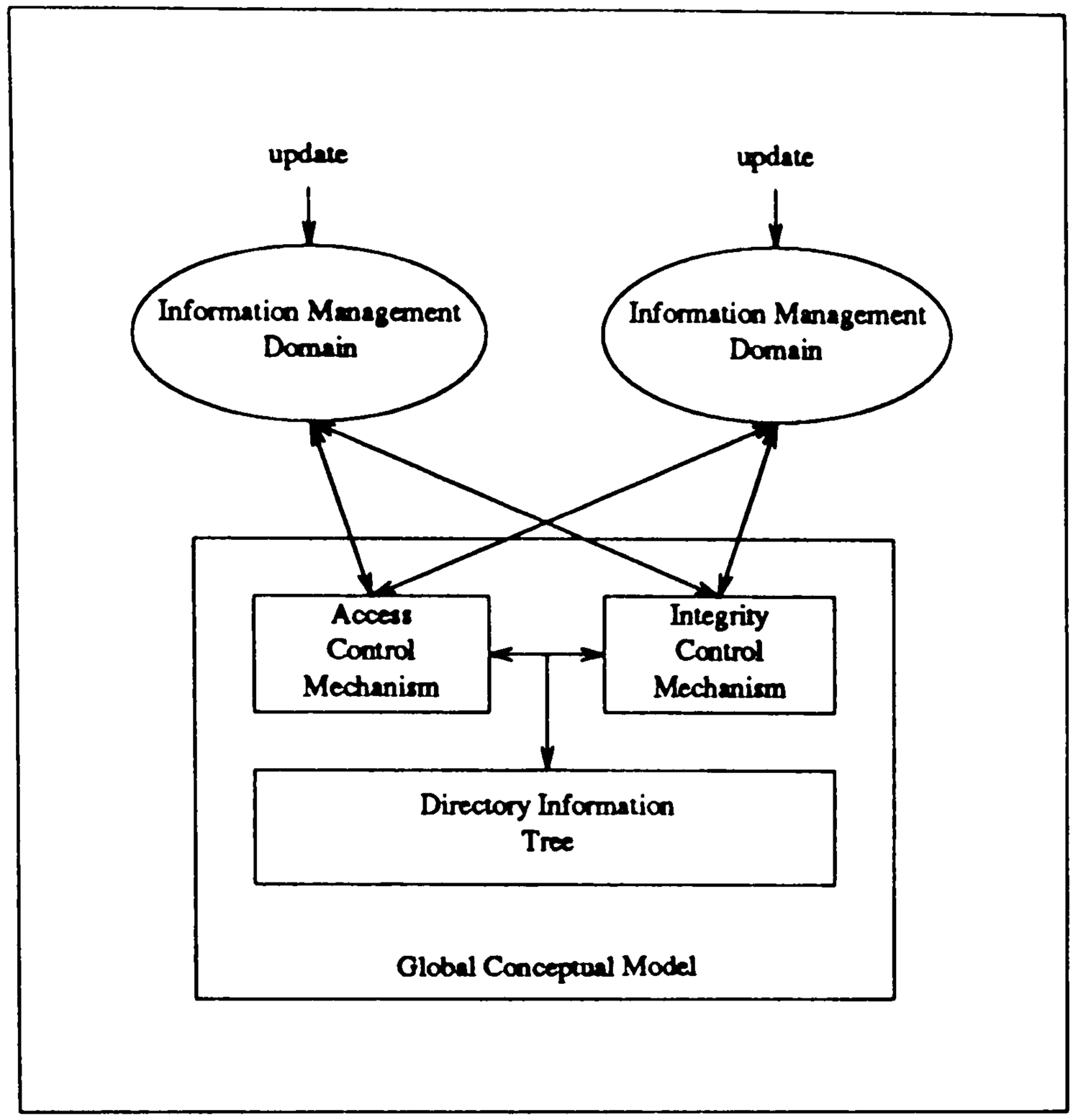


Figure 4.1: Many Information Management Domains supported by the global security and integrity mechanisms

It should be noted that Information Management Domains are not equivalent to *Directory Management Domains* (DMDs) as defined by X.500 [CCITT-X501]. IMDs represent logical boundaries for information management whereas DMDs represent boundaries for Directory system management. These need not be the same, although they may often be related.

This section has noted that the management of information is a complex subject and has defined the interests of this thesis in terms of the specific issues of access control and data integrity. It has also described the concepts of dynamic management and Information Management Domains affecting the design of the access control and data integrity mechanisms. The following section turns its attention to the problem of access control.

4.2. An access control mechanism for the Directory service

This section describes a data access control mechanism for the Directory service. An overview of general access control theory is presented in section 4.2.1 followed by the specific requirements of directory access controls in section 4.2.2. These are used to specify a directory access control mechanism based on *Access Control Lists* (sections 4.2.4 to 4.2.10) paying particular attention to the flexible description of actions and groups of directory users.

The overall security of directory information may involve a number of issues ranging from the physical security of buildings containing computing equipment to the security of underlying storage and communication services. An annex to part X.518 of X500 (*Authentication Framework*) describes a number of issues relating to directory security such as *data confidentiality*, *non-repudiation* and *access control*. This section considers the last of these, namely, controlling access to directory information at the global conceptual layer. This requires the specification of an access control mechanism controlling the use of abstract operations to read and manipulate the Directory Information Tree. Two major motivations for providing this access control mechanism are as follows:

- The need for organisations and individuals to make only a subset of their directory information available to other users. This requires constraints on who can read, browse, or even know of information.
- The need for the Directory service to reflect the update policies of different Information Management Domains. This requires constraints on who may add, delete or modify directory information.

These motivations may be summarised by the words *privacy* and *administration*.

Access control mechanisms have been the topic of extensive study within the context of general databases and operating systems [DEM78, SIR86, FER81, TAN87, DATE77] and the following section provides an overview of this work in order to identify the major problems to be addressed by the directory access control mechanism.

4.2.1. An overview of general access control

Access controls are rules governing which entities may perform which operations on what information or resources. They therefore represent a binding between users, objects (information/resources) and operations. This binding may be expressed as a matrix called the *Authorisation Matrix* in which rows correspond to users, columns correspond to objects and elements contain descriptions of operations. An example Authorisation Matrix is shown in figure 4.2.

	object 1	object 2	. . .	object j	. . .	object n
user 1						
user 2		Read				
. . .						
user i				Read Add,Del		
. . .						
user m		Add,Del Read,Mod				Read Mod

Figure 4.2: Example Authorisation Matrix

The element *A[i,j]* of the Authorisation Matrix represents the operations which user *i* may perform on object *j*. A user action may be described by a set of requests of the form *R[i,j]* indicating the operations to be performed by the user (*i*) on an object *j*. For each request, a system process called the *Arbiter* checks the Authorisation Matrix and determines whether the operations *R[i,j]* are included in *A[i,j]*. If they are, access permission is granted; if not, it is denied. Denial may have several consequences ranging from returning a suitable message to the user to logging the user off and informing the relevant authorities.

Access Rights may be granted and revoked by altering the Authorisation Matrix. This also occurs subject to access control thus requiring the matrix itself to be an object in the system.

Large systems often contain many users and objects with each user having access to only a relatively small number of objects. The resulting Authorisation Matrix is therefore large and sparse and it may be inconvenient to store or view it as a whole. It is also difficult for a distributed system to implement a single Authorisation Matrix which would either require a centralised access control mechanism resulting in a loss of nodal-autonomy or many copies of the matrix at different nodes generating consistency problems.

As a result, the Authorisation Matrix is often split up and the resulting fragments distributed around the system. Splitting generally occurs in one of the following ways:

- The matrix may be split row-wise into a number of elements representing the access rights of specific users in terms of operations and objects. Each row is a user's *Capability List* and each element of a row is a *Capability* granting certain types of access to

a named object.

- The matrix may be split column-wise into a number of elements representing all access rights granted for specific objects. Each column is an *Access Control List* (ACL) for an object and each element of a column is an *Access Control List Element* granting certain types of access to a named user or group of users.

Although Capabilities and Access Control Lists contain the same functionality, they have different side effects on the systems implementing them, resulting in significantly different flavours of access control. Both ACL and capability based systems are in use today with the UNIX operating system being an example of the former [WAL85] and the Amoeba operating system being an example of the latter [MUL87]. The splitting of the Authorisation Matrix is shown in figure 4.3. Capabilities and ACLs are described in more detail in the following sections.

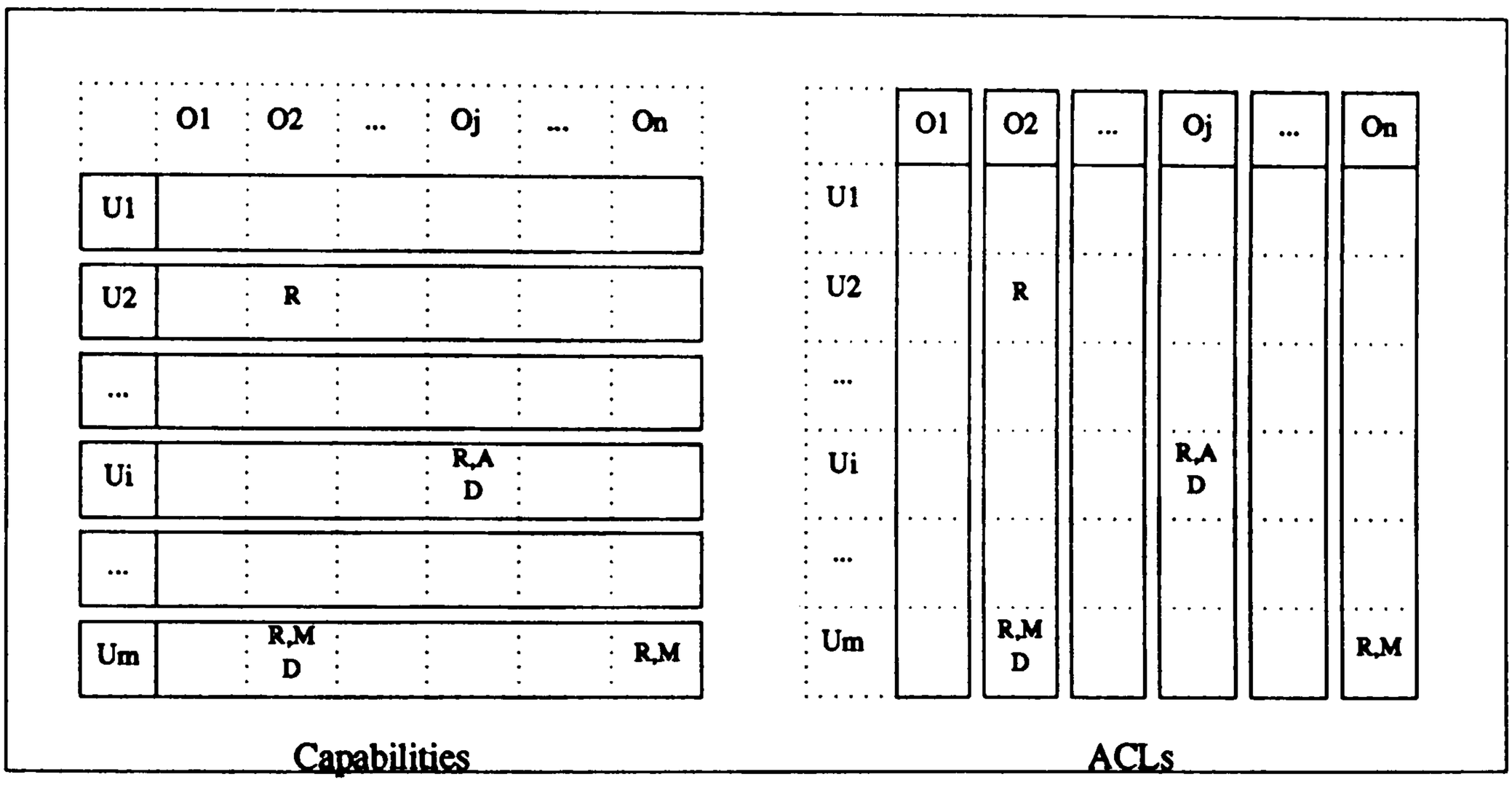


Figure 4.3: Splitting the Authorisation Matrix into Capabilities and ACLs

Capabilities

Capability mechanisms represent access control information by Capability Lists associated with each user. Capabilities can be viewed as tickets, presented to the Arbiter for checking whenever a user wishes to operate on some information. Capabilities are granted to users by a *capability server* and need to be encrypted for security reasons. However, once granted, the Capabilities may confer the right for the user to transfer or copy them to other users. If

properly controlled this is a useful feature of a capability system.

A distributed system requires that the capability server is itself distributed and, furthermore, a user's Capabilities are often stored at the node holding the user's information, as opposed to the node holding the information to be accessed. It follows that all of the access controls relevant to a piece of information may be stored at many nodes in the system making it difficult for the owner of information to determine which users have access to it. Consequently, it is difficult to support the selective revoking of access rights using a distributed, capability based access control mechanism (although the blanket revoking of all access rights for an object is possible).

A capability mechanism in a large distributed system may require each user to have a large Capability List (even with a sparse mapping from users to objects) and in order to make these lists manageable a mechanism to concisely describe sets of information is required.

A further, important feature of Capabilities is that it is not strictly necessary for a user to give their name when performing an operation so long as the system is able to determine that the Capability is valid. Thus, it is possible for a user to retain anonymity when using the system. This may be viewed as useful or dangerous depending on the ethics of system users and providers.

In general, it can be seen that capability based access control mechanisms shift the balance of power from administrators to users. Once Capabilities have been issued, it is difficult to selectively revoke them and to trace the actions of their owners who are therefore entrusted with greater responsibility.

Access Control Lists

Access Control List (ACL) based mechanisms represent access control information in the form of lists of users and operations associated with each piece of information. In requesting an operation the user supplies some identification (usually their name) which is then compared with the relevant ACL to determine whether they have the necessary access rights.

In a distributed system, ACLs are usually stored at the same node as the information they protect making it easy for the information owner to determine which users possess which rights for the information, and to selectively revoke rights if necessary. The transfer of access rights from one user to another requires the modification of ACLs. This means that Access Control Lists are susceptible to user name changes which may invalidate them resulting in a loss of all of rights for the user concerned.

In a system with large numbers of users the access control mechanism requires a method of concisely describing groups of users in order to facilitate the management of ACLs.

Furthermore, a method describing groups of users in generic terms may accommodate name changes more easily. For example, an ACL conferring access rights to all members of an organisation would be unaffected by a user name change provided the user remained within that organisation.

In general, ACL based mechanisms concentrate power in the hands of the information owners and administrators who know which users have which rights and are able to selectively grant and revoke rights.

Authentication

Authentication is the process of identifying a user to the satisfaction of another user or system and is required if an access control mechanism is to operate successfully. The capability based mechanism requires that the capability server is able to authenticate a user before granting Capabilities and the ACL based system requires that the Arbiter authenticates a user before checking their access rights against an ACL.

Authentication and access control can be seen as orthogonal issues. Authentication in an Open Systems environment has been tackled by other work [CCITT-X518] and is beyond the scope of this thesis. The remainder of this chapter assumes the existence of a separate *Authentication service*, called by the Directory service to authenticate users at will.

4.2.2. Requirements of the directory access control mechanism

This section begins the specification of the directory access control mechanism by describing its major requirements. The Directory service is a specialised distributed database operating under a set of constraints strongly influencing its design. Consequently, the directory access control mechanism may differ from mechanisms within more general database or operating systems.

Flexibility to support different Information Management Domains

Section 4.1 described the role of *Information Management Domains* in representing different access control and data integrity policies. Access control policies may vary greatly and the directory access control mechanism must be flexible enough to support a wide range of approaches at the external level. For example, one IMD might define simple access controls in terms of two classes of user, a *directory administrator* having rights to all information and other *users* having read and browse rights only. Alternatively, a second IMD might allow all users to create information, and then the information creator to control who could read and update the information.

The flexibility required of the directory access control mechanism manifests itself in three main areas:

High granularity of information

Access controls should apply at a high granularity of directory information. If necessary, the Directory should support separate access controls for individual attributes within an entry.

High granularity of actions

Directory abstract operations such as *Modify Entry* may include a number of logical sub-operations at any one time. These sub-operations should be protected by different access rights to provide sufficient flexibility in describing actions. This may lead to a more complex global conceptual model. However, simplification may occur at the external layer.

Describing groups of users

Unlike many databases, the Directory operates on a world wide scale and the naming of users is of prime importance. The directory access control mechanism must control large numbers of users and therefore requires a concise, flexible method for representing groups of names. This method should allow the assignment of access rights reflecting natural organisational access groupings and it should also be possible to group users by generic properties such as titles and roles.

There are three other major requirements for the directory access control mechanism:

Support both hidden and forbidden information

The denial of access to information could take two general forms: the Arbiter may inform the user that they do not have access permission for the requested information or, alternatively, it may act as if the information did not exist at all. In the first case, the information is *forbidden* and the user is aware that it exists but is unable to access it. In the second case, the information is *hidden* and the user cannot tell whether it exists at all. The directory access control mechanism should support both hidden and forbidden information depending on the security policy in force.

Dynamic update of access controls

It should be possible to dynamically alter access control information, via the Directory Access Protocol, to reflect new access policies and therefore support dynamic management as described above. This implies that the Directory must provide access controls for its access control information.

Guaranteed access

The access control mechanism must prevent situations where information is in a state of deadlock because no user has permission to alter or delete it. This requirement does not present an obvious problem with centralised systems where it is easy to keep track of users. However, in the case of the globally distributed Directory it may be difficult to determine whether access controls allow any access at all.

4.2.3. Reasons for choosing an ACL based mechanism

Section 4.2.1 above provided an overview of both capability and ACL based access control mechanisms. Each approach has its advantages and both may be applicable to the Directory service in the long term. However, the immediate future favours the ACL based approach for the following reasons:

- Capabilities may require greater support from a general distributed security framework. For example, they require the use of encryption services to prevent the creation of fake capabilities. At the present time, the necessary security framework remains the subject of basic research and is therefore not generally available. It should be noted that the adoption of an ACL based access control mechanism does not escape the need for encryption techniques. However, under the ACL approach, encryption is required within the authentication service, not necessarily within the Directory itself.
- In the short term, the Directory will be provided by a small number of powerful organisations such as the national PTTs. These organisations will most likely wish to retain tight control over information and administrative power will therefore be in the hands of information providers as opposed to users.

In the long term, a capability based mechanism may prove to be equally, or even more, applicable to the Directory Service. One important reason for this may be the *anonymity* which capabilities can provide for directory users.

The directory access control mechanism specified by this thesis is based on Access Control Lists as opposed to Capabilities due to their short term suitability. However, it should be noted that this mechanism alone may not provide the only long term solution.

The remainder of section 4.2 describes the extension of the directory information model and abstract operations to support an access control mechanism, based on Access Control Lists, meeting the requirements of section 4.2.2. Section 4.2.4 provides an overview of the directory access control mechanism. Sections 4.2.5 and 4.2.6 describe the structure of ACLs in detail. Finally, sections 4.2.7 to 4.2.10 consider public directory access, avoiding access deadlock and manipulating ACLs via directory operations.

4.2.4. Extending the information model to include ACLs

This thesis specifies a directory access control mechanism using Access Control Lists (ACLs) to protect information. This work is based on a model from the ISO/CCITT Melbourne X.ds output [CCITT-XDS86] which is extended to provide a detailed and flexible solution.

An ACL is a structure associating sets of actions with groups of users and is interpreted as allowing the users to perform the actions. The term *access rights* is used throughout the remainder of this chapter to refer to the actions permitted to a group of users by an ACL.

Each entry in the Directory Information Base may be protected by an *entry level ACL* assigning access rights for those actions affecting the entry as a whole such as deletion or suspension. Each attribute type within an entry may be protected by its own *attribute level ACL* assigning access rights for attribute specific actions such as addition or modification of a value.

Both entry and attribute level ACLs are optional, with the entry level ACL acting as the default for all attributes not having their own attribute level ACLs. This default rule increases the efficiency of the mechanism by allowing entry level ACL to describe general access rights for the entry as a whole but allowing them to be overridden by specific attribute level ACLs when necessary. An entry without an entry level ACL (i.e. an empty entry level ACL) is considered public (see section 4.2.7) except for those attributes specifically protected.

The extended structure of an entry supporting ACLs is shown in figure 4.4.

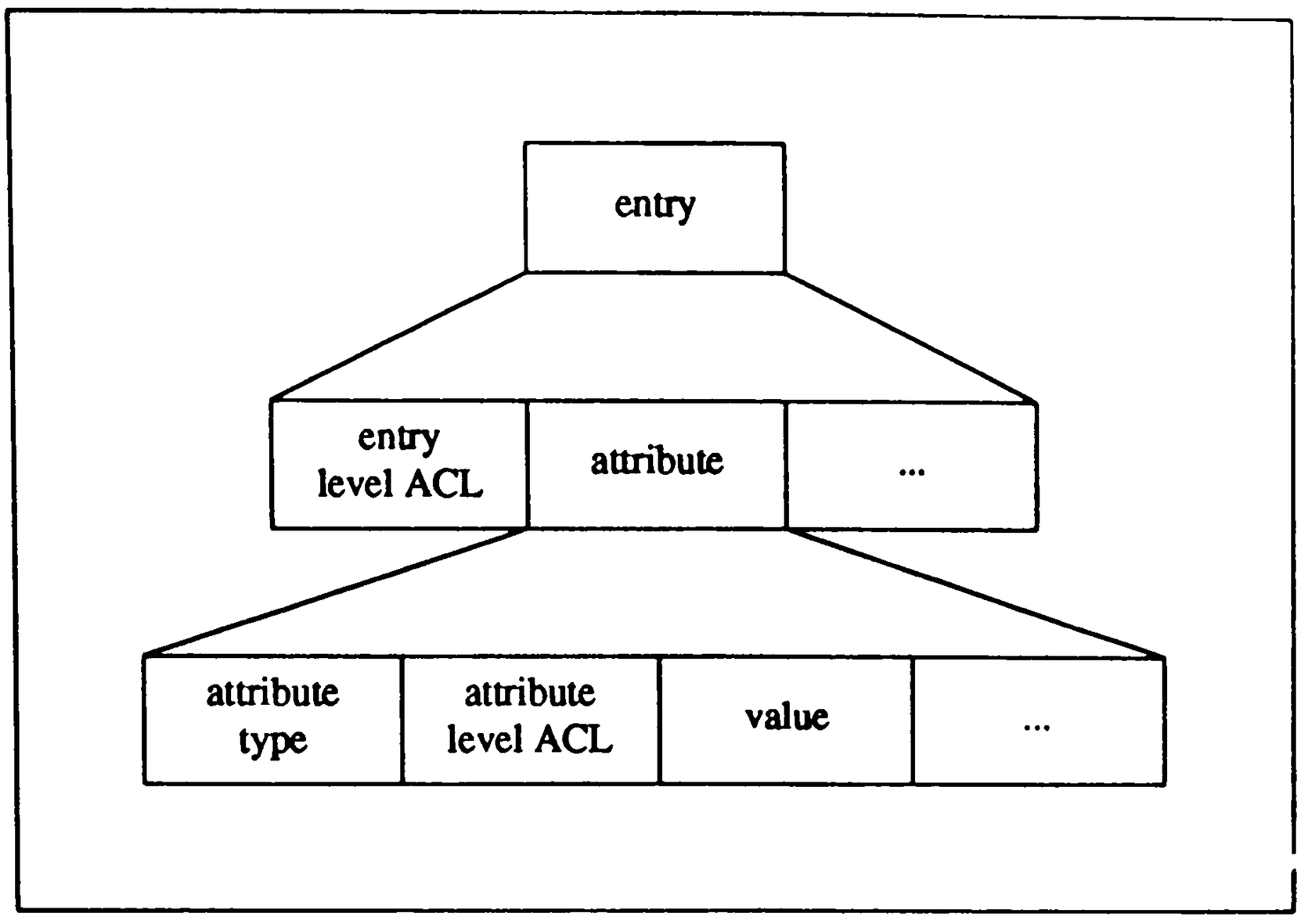


Figure 4.4: Structure of an entry including ACLs

The overall operation of the directory access control mechanism is as follows. Each operation includes the name of the user requesting it. The operation may access several attributes and entries simultaneously and can be broken down into a set of logical requests accessing individual entries or attributes. For each such request, the access control mechanism finds the relevant ACL (using the default rule for attributes) and checks whether it assigns the necessary access rights to the user. If so, access is granted. If not, it is denied. The checking of ACLs for access rights is summarised by the following procedure.

```
For each entry specified within the operation {

        Return a name error if the entry level ACL
        "hides" the entry.

        Deny access if the entry level ACL does not give permission
        for actions affecting the whole entry

        Otherwise, grant general access to the entry

        For each attribute to be operated on {

                Use the attribute level ACL if it exists. Otherwise
                use the entry level ACL (default rule).

                Indicate an attribute error if the ACL "hides" the
```

```
                    attribute.

                    Deny access to this attribute if the ACL does not
                    give permission for the requested actions

                    Otherwise, grant access to the attribute
                }
        }
```

Denial of access has different effects for different operations. Generally, read operations return a combination of results and errors corresponding to information for which they were granted or denied access. However, due to the *one out - all out* transaction control policy (see section 3.3.3), update operations abort and return an error at the first point where they are denied access.

The previous paragraphs have broadly outlined the operation of the access control mechanism in terms of entry and attribute level ACLs. In order to further specify the directory access control mechanism, it is necessary to examine the structure of ACLs in greater detail and, in particular, how they represent actions and users.

An ACL can be modelled as a set of *Access Control List Elements* (ACL Elements) each of which assigns a set of access rights to one particular group of users. The remainder of this chapter refers to a group of users, assigned some access rights, as an *access group*. The basic structure of an ACL is shown in figure 4.5.

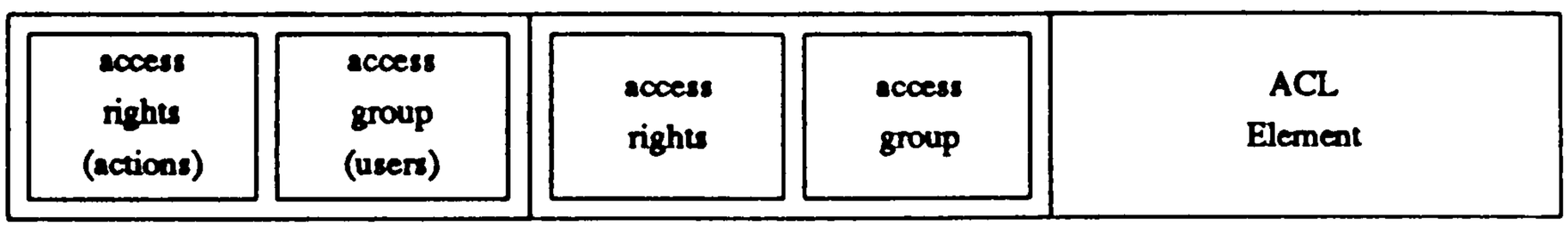


Figure 4.5: The overall structure of an Access Control List

The following sections discuss the structure of ACLs in greater detail concentrating on the following issues:

How access rights are represented within ACL Elements (section 4.2.5).

How access groups are represented within ACL Elements (section 4.2.6).

4.2.5. The structure of ACLs: representing actions

This section specifies a flexible mechanism for representing access rights within ACL Elements.

Each ACL Element must be able to represent the full range of logical actions which can be applied to the protected item of information. A first approach might be to represent actions by the names of abstract operations. Each ACL Element would then associate a list of operations with a specific access group. For example an ACL might consist of two elements assigning the access rights *(Read Entry, Modify Entry, Delete Entry)* to one access group and the access rights *(Read Entry, Search)* to another. However, simply listing the names of operations is subject to the following problems:

- A single abstract operation may group together a number of logical operations for reasons of efficiency and transaction control. A good example is a *Modify Entry* operation, simultaneously adding a new attribute, deleting an old attribute and replacing a third attribute value. Describing actions by the names of abstract operations gives too coarse a granularity of access control.
- Describing actions by the names of abstract operations does not allow the access control mechanism to distinguish between hidden and forbidden information.

A better approach is to describe actions in terms of *access categories* referring to the lower level logical actions applying to information. Permission to perform an operation might require a number of different access categories. Access categories must distinguish between permission to know of and read information; permission to add, delete and replace attributes; permission to add and delete entries and aliases; permission to suspend and reinstate entries and finally permission to modify ACLs themselves.

The table below lists an initial set of access categories and outlines their purpose. This set will expand as new abstract operations are introduced in later work.

Access categories	
category	meaning
detect	Allows knowledge of the existence of information
read	Allows attributes to be read
add_value	Allows a new attribute value to be added
delete_value	Allows an attribute value to be deleted
replace_value	Allows an attribute value to be altered
add_entry	Allows a new (child) entry to be created
delete_entry	Allows an entry to be deleted, suspended or reinstated
alias	Allows a new alias to be created
reset_alias	Allows an alias to be reset
read_ACL	Allows Access Control Lists to be read
replace_ACL	Allows the modification of an ACL

The following comments apply to this table:

- The *detect* access category is used to distinguish between hidden and forbidden information. If the detect category is not granted the Directory behaves as if the specified entry or attribute did not exist and returns name or attribute errors. Lack of other access categories results in access control errors being returned to the user.
- There are no specific access categories dedicated to the creation and deletion of ACLs because these actions are achieved by the creation and deletion of attributes or entries and are therefore governed by existing access categories.
- The access control mechanism distinguishes between creating entries and aliases by the *add_entry* and *add_alias* categories at the relevant naming authority. However, once created, only a single *delete_entry* category is required at the target entry or alias.
- Suspending an entry involves its eventual deletion and is therefore covered by the *delete_entry* access category.

The table below shows the access categories required during the course of each operation.

Access categories required by each operation	
operation	categories
Read Entry	detect, read, read_ACL
List Children	detect
Search	detect, read, read_ACL
Modify Entry	detect, add_value, delete_value, replace_value, replace_ACL
Add Entry	detect, add_entry
Delete Entry	detect, delete_entry
Add Alias	detect, add_alias
Delete Alias	detect, delete_entry
Suspend Entry	detect, delete_entry
Reinstate Entry	detect, delete_entry
Reset Alias	detect, reset_alias

In summary, we can represent actions within ACLs by combinations of access categories representing logical operations on information. An example ACL using access categories is shown in figure 4.6

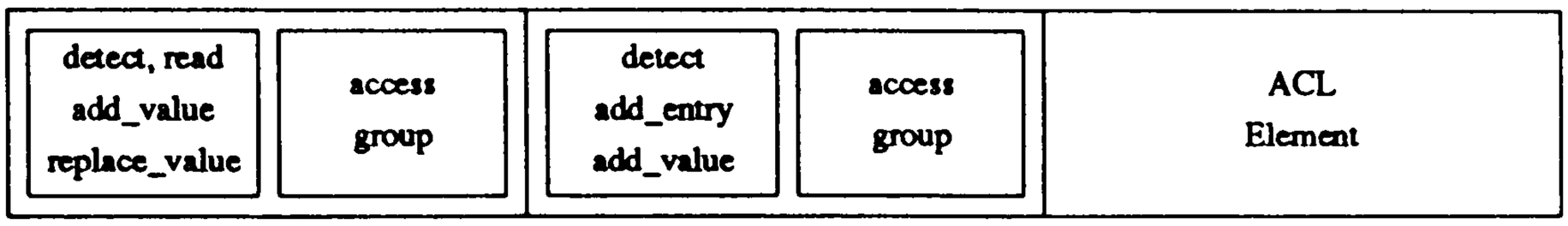


Figure 4.6: Example Access Control List using access categories

4.2.6. The structure of ACLs: representing access groups

This section completes the structure of ACLs by specifying a mechanism for representing access groups (groups of users).

The simplest method of describing groups of users would be to list their distinguished names. However, this suffers from the following drawbacks:

- Lists of names may become long and unmanageable in a global Directory service.
- Lists of names describe a flat user space and fail to exploit natural access groups occurring within the real world.
- Lists of names are static and do not easily accommodate changes to the Directory Information Tree. This is a general problem with access control mechanisms based on ACLs where a user name change may invalidate many ACLs throughout the system.

These problems highlight the requirement for a concise, flexible method of describing access groups within ACLs. Before specifying a solution to these problems, it is necessary to make an observation concerning likely access groups within the real world.

> Directory access groups are determined by the security policies of different Information Management Domains. Although these may vary, the following work assumes that IMDs generally reflect organisational structure and that access groups are often based on organisational hierarchy. This is really an extension of the assumption made when developing the Directory Information Tree. Examples of hierarchical access groups are the members of an organisation, department, project, working group or office.

Representing users by Object Set Descriptors

The following sections describe a structure called the *Object Set Descriptor* (OSD) used to represent sets of distinguished names. OSDs are adopted to represent access groups within ACLs.

The set of names described by an OSD is called its *object set*. Two logical operations may be applied to an OSD:

- *Enumeration* maps from the Object Set Descriptor to its object set.
- *Verification* determines whether a specified name belongs to the object set of an OSD.

An Object Set Descriptor consists of two parts: a *subtree* reflecting an organisational access group and a *filter* further refining the access group. The use of subtrees and filters is described below.

Use of subtrees

The Directory Information Tree represents organisational hierarchy and, under the above assumption, represents natural user access groups. It follows that groups of users can be represented by subtrees of the DIT. Subtrees form the basis of Object Set Descriptors.

In addition, it is useful to limit the depth of subtrees representing access groups. Object Set Descriptors therefore allow subtrees to have a specified depth in terms of a number of levels. Unlimited depth is assumed when unspecified.

Subtrees may be represented by the distinguished names of their root entries. For example, we could represent all members of Nottingham University by an Object Set Descriptor specifying the subtree with root */C=GB /O=Nott.Uni/* and with unlimited depth. It should be noted that a list of users' distinguished names is a trivial case of a list of subtrees.

Use of filters

Access groups based on subtrees of the DIT give a useful, but coarse, method of assigning access rights to users. Access controls may also be granted on the basis of specific user characteristics such as roles, titles and status. For example, one may wish to assign certain access rights to system administrators or employees of a specified grade. These generic access groups may be represented by a *filter* as described in section 3.3.2. A filter is an expression based on logical combinations of attributes which are matched against entries in the DIT. Thus, entries representing system administrators might match the filter *class = person AND role = system administrator*. An Object Set Descriptor may therefore include a filter further constraining the entries in a directory subtree.

The final structure of an OSD is given by a distinguished name representing a DIT subtree, constrained to be of a certain depth, associated with a filter describing specific properties of entries in the object set. This structure is described by the following ASN.1 code.

```
ObjectSetDescriptor ::= SET {
        subtree [0] DistinguishedName,
        level [1] CHOICE {
                fixed [0] INTEGER,
                unlimited [1] NULL DEFAULT },
        filter [2] Filter OPTIONAL }
```

The use of Object Set Descriptors reduces the problems associated with flat lists of names in the following ways:

- Naming a subtree is more concise than listing all the names in a subtree.
- OSDs represent natural access groups by a combination of subtrees and filters.
- OSDs may be unaffected by minor name changes so long as a user remains within the specified subtree and continues to satisfy the filter. This reduces the chances that ACLs will be invalidated by name changes.

The following are examples of Object Set Descriptors representing access groups.

> All postgraduate students within the department of Computer Science at Nottingham might be represented by the OSD with subtree */C=GB/ O=Nott.Uni /OU=CS/* (unlimited depth) matching the filter *title = postgraduate student*.
>
> The managing directors of the company "Widget International" could be represented by the OSD with subtree */C=USA /O=Widget International/* (unlimited depth) matching the filter *title = managing director*.

Object Set Descriptors are similar to the arguments of *Search* operations. A search can be viewed as the enumeration of OSDs whereas checking access controls requires the

verification that an entry satisfies an OSD.

The use of OSDs provides a flexible method of describing access groups. However, it does not allow public access to directory information. An extension to support this functionality is considered in the next section.

4.2.7. Public access to information

One implication of the use of OSDs for describing access groups is that users of the Directory service must be explicitly represented within the DIT. However, there may be occasions when non-registered users wish to use the service or when registered users wish to act anonymously. In the first case, a user does not have a directory name and in the second case they do not wish to supply one. This may be supported by introducing the concept of public directory access. A special *public* dummy access group may be introduced including any entity accessing the Directory. This access group may be assigned access rights just as any other, enabling the Directory to give limited public access to information on a flexible basis. For example, an ACL might assign the detect and read rights to the public access group.

The Directory must also act when information is unprotected due to the absence of both entry and attribute level ACLs. A sensible approach is to assume default access controls allowing the public to read and browse but not to update information (i.e. assume detect and read access categories for everyone).

4.2.8. The final structure of ACLs

The previous sections have specified a directory access control mechanism based on the use of Access Control Lists to protect entries and attributes within the DIT. This section briefly summarises this work.

ACLs specify which actions may be applied to directory information by which groups of users. Actions are referred to as *access rights* and groups of users are referred to as *access groups*. ACLs may be associated with entries and attributes and a default rule applies between these two levels. Each ACL consists of a set of elements assigning access rights to different access groups where:

- Access rights are described by sets of access categories.
- Access groups are described by Object Set Descriptors.
- A special public access group allows anonymous users to have limited access to directory information.

Access categories provide great flexibility in describing actions and Object Set Descriptors are a concise and dynamic way of reflecting real world access groupings. Access Control

Lists are defined by the following ASN.1 description:

```
AccessControlList ::= SET OF AccessControlListElement

AccessControlListElement ::= SEQUENCE {
                categories BIT STRING { detect(0), read(1), etc. },
                accessgroup CHOICE {
                        public [0] NULL,
                        users [1] ObjectSetDescriptor } }
```

The final structure of Access Control Lists is indicated by figure 4.7 below. This shows an example ACL assigning detect and read rights to the public and detect, read, add_value and replace_value rights to all administrators within Nottingham University.

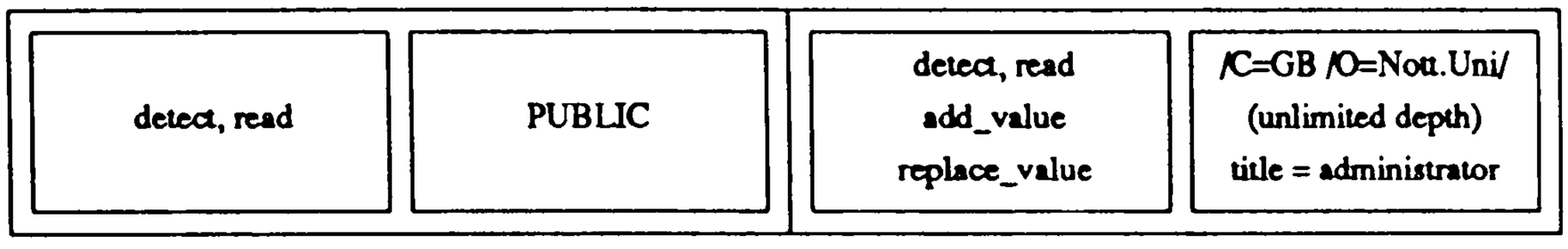


Figure 4.7: An example Access Control List using Object Set Descriptors

The following sections discuss two remaining issues concerning access controls. The first considers the problem of dynamically manipulating ACLs and the second examines the problem of avoiding deadlock due to invalid or meaningless ACLs.

4.2.9. Granting, revoking and managing access controls

The previous sections have specified the structure of Access Control Lists for the protection of directory information. In order to support dynamic management, the Directory must allow the dynamic creation, update and removal of ACLs enabling access permissions to be granted or revoked as users and policies change. This can be achieved by extending the abstract operations specified in section 3.3 in the following ways:

- The ACLs belonging to an entry may be retrieved along with its attributes by extending the *Read Entry* and *Search* operations to include an optional flag requesting the return of ACLs.
- The addition of ACLs occurs with the addition of new entries and attributes. Thus, ACL addition can be supported by extending the *Add Entry* and *Modify Entry* operations to include ACLs within their arguments.

- The modification of ACLs can be achieved using an extended *Modify Entry* operation specifying replacement ACLs as well as replacement attributes.
- An ACL is deleted by replacing it with an empty ACL. This uses the extended *Modify Entry* operation described above.

In general, permission to read an ACL requires the *read_ACL* access category and permission to replace an ACL requires the *replace_ACL* access category, both within the target ACL.

The above extensions to the Directory Access Protocol supporting the retrieval, creation and maintenance of ACLs are fully specified in ASN.1 form in appendix A of this thesis.

4.2.10. Guaranteed access to information

In order to avoid access deadlock, at least one entity must be guaranteed access to each piece of information. The problem of deadlock due to invalid access control is particularly relevant to the Directory because of its large user name space. To illustrate the problem, consider an entry level ACL assigning update rights to users represented by an Object Set Descriptor. Over a period of time these users may be deleted from the DIT rendering its object set empty. At this point there will be no entity who can satisfy the ACL and therefore delete the protected entry. Consequently, no one will be able to delete any of its superior entries because only leaf entries can be deleted. This deadlock situation is highly undesirable.

There are two approaches to solving this problem:

1. Ensure that every ACL in the Directory always assigns rights to at least one valid user and prevent the DIT from being altered to contradict this.
2. Designate privileged directory super-users with the power to remove information, thus breaking any access deadlock.

The first of these solutions is not practicable due to the size and distributed nature of the DIT. It would require checking that all directory ACLs would remain valid before an entry could be deleted. It would also require the enumeration of ACLs whenever they were altered. This would be ridiculously expensive.

The second solution is feasible although great care has to be taken that a directory superuser does not represent a security loophole. This thesis adopts the second solution and proposes that each DSA in the Directory system supports a *DSA administrator* who has super-user powers for all information stored by that DSA. The DSA administrator is local to a single DSA and is not visible to the users of the Directory. In addition, they do not have any distributed access to information. This increases security because the lack of superuser network

access between DSAs makes the system harder to break and limits the damage which can occur if it is broken.

This concludes the specification of the directory access control mechanism. The remainder of this chapter specifies an integrity mechanism supporting the integrity aspect of information management.

4.3. A data integrity mechanism for the Directory service

The goal of this section is to specify a directory data integrity mechanism. This mechanism facilitates the correct and meaningful update of information by allowing the definition of integrity rules, enforced during update operations. The structure of directory integrity rules is specified as well as new abstract operations supporting their dynamic definition and hence the ideal of dynamic management. The integrity mechanism also reflects the concept of Information Management Domains.

- Section 4.3.1 presents the motivations for the directory data integrity mechanism.
- Section 4.3.2 gives a brief overview of the integrity mechanism and describes the role of entry and attribute definitions.
- Section 4.3.3 specifies the structure of attribute definitions.
- Section 4.3.4 specifies the structure of entry definitions.
- Section 4.3.5 examines support for Information Management Domains.
- Section 4.3.6 examines support for dynamic management.
- Section 4.3.7 presents a summary of the integrity mechanism and some examples of its use.

Finally, section 4.4 closes chapter 4 with a review of the directory global conceptual schema and its extension to include access control and data integrity mechanisms supporting the management of information.

4.3.1. Goals and motivations of the integrity mechanism

Integrity is concerned with maintaining the correctness of information against invalid alteration or destruction and is supported by a *data integrity mechanism* allowing users to define rules governing both the structure and contents of information. The integrity mechanism then ensures that these rules are adhered to during updates.

The following paragraphs describe two major motivations for specifying a directory integrity mechanism.

Defining the structure of information

The first motivation concerns the structure of information. Many applications using the Directory expect to find information in a well known state and cannot function otherwise. They require that names adhere to well defined forms, that entries contain certain attributes and that attributes have sensible values. For example, a distribution list application might expect distribution list entries to contain the attributes *submitting member*, *receiving member* and *moderator* and would expect the values of these attributes to be the names of communication entities as opposed to numbers or bit strings.

The directory integrity mechanism must therefore allow users to specify the structures of name-forms, entries and attributes and must ensure that these structures are preserved as information is updated.

Supporting forms based user interfaces

The second motivation concerns support for user interfaces.

Humans often make direct use of the Directory service. They may wish to obtain information describing services and other communication entities or they may be acting in an administrative role and maintaining information. In either case, they require extensive support from a user interface so that they can effectively search and manipulate the Directory Information Tree. For example, the creation of a directory entry might be aided by a form prompting the user for the values of suitable attributes.

Another important example is the support required for *user friendly* naming. Section 1.3.2. made the point that user friendly naming will probably involve dialogue between humans and the Directory. This dialogue may take the form of the user interface suggesting possible naming attributes to the user who responds with suitable values in an attempt to gradually resolve a name.

Both of these examples could be supported by a *Query by Example* style interface [ZLO75] where the user interacts with the Directory through a series of templates for names, entries and attributes. In order to support such an interface, the DUA itself must understand the structure of directory information. This requires the presence of meta-information defining the directory information structure and therefore provides another motivation for the directory integrity mechanism. This example indicates that, by defining the structure of directory information, the integrity mechanism is defining the *global conceptual schema*.

In addition to the general aims of supporting the correct update of information and sophisticated user interfaces, the data integrity mechanism should satisfy the following specific requirements.

- It should support the dynamic definition and management of integrity rules as part of the Directory Access Protocol. This means that it should be possible to create and remove integrity rules using abstract operations and not via some extra-directory mechanism.
- The integrity mechanism should allow different Information Management Domains to specify their own integrity rules according to local policies. This suggests the need for integrity rules to have a *scope* constraining their effect to specific areas of the DIT representing the boundaries of IMDs.
- The integrity mechanism should support the inheritance of integrity rules between IMDs. It should allow the local modification of rules while still remaining compatible with more global definitions.

The following sections specify a directory integrity mechanism to meet the goals described above. This mechanism supports the dynamic definition of directory name-forms, entries and attributes. It pays particular attention to the concept of *scope* for integrity rules.

4.3.2. Overview of the directory integrity mechanism

This section provides an overview of the proposed data integrity mechanism. Later sections describe specific aspects of this mechanism in greater detail.

The data integrity mechanism defines the structure and contents of names, entries and attributes via two mechanisms called *attribute definitions* and *entry definitions*.

- *Attribute definitions* define the abstract structure of attributes by assigning attribute types, specifying the basic structure of values and placing constraints on values.
- *Entry definitions* define the abstract structure of entries and specify possible directory name-forms. They describe the attributes which are mandatory or optional within a class of entry and may specify default values and access controls for these attributes. Entry definitions also constrain the hierarchical relationships between entries and hence describe possible directory name-forms.

The integrity mechanism includes new abstract operations for creating and removing attribute and entry definitions and specifies extensions to the directory access control mechanism to control their use. The mechanism also expresses the effects of attribute and entry definitions on the existing abstract operations in terms of the conditions they must obey.

Each attribute and entry definition is declared within a *scope* limiting its area of effect to a specified subtree in the DIT. The concept of scope allows the integrity mechanism to support the policies of different external Information Management Domains.

Attribute definitions are fully specified by section 4.3.3 and entry definitions are specified by section 4.3.4.

4.3.3. Attribute definitions

An attribute definition declares the abstract structure of a type of attribute to the Directory. Each attribute within the Directory must correspond to an attribute definition specifying its name, scope and structure as well as placing constraints on its values. An attribute definition therefore describes a class of attributes adhering to it. This is called an *attribute class*.

4.3.3.1. The structure of an attribute definition

The ASN.1 structure below specifies an attribute definition. The elements of this structure are described in the following paragraphs.

```
AttributeDefinition ::= SET {
        type [0] PrintableString,
        scope [1] DistinguishedName,
        recurring [2] BOOLEAN,
        structure [3] AttributeStructure (see below) }
```

The *type* has the function of an *attribute type* as used throughout the previous chapters. It is the handle by which most directory users refer to an attribute and is structured as a human readable character string.* Examples of attribute types are *title*, *common name* or *moderator*.

The *scope* defines a subtree of the DIT in which this attribute definition is valid. This means that attributes for this definition can only exist within this subtree. The scope is represented by the distinguished name of the root entry of the specified subtree. Scope defines the locus of effect of an attribute definition and its purpose is discussed in section 4.3.5.

The *recurring* flag indicates whether an instance of this attribute may have more than one value within a specific entry. This basic integrity check ensures that some attributes such as *passwords* or *serial numbers* are guaranteed to be single valued.

---

*This is different to the structure of attribute types within X.500

The *structure* describes the syntax of the attribute value and may place some constraints on its contents. An attribute value may be structured as one of several basic data types or as a directory *name*. The basic data types are derived from the primative data types specified in ASN.1 and are *integer*, *real*, *IA5string*, *bitstring*, *octetstring*, *boolean* and *generalised time*.

Numeric values may be constrained by a maximum and minimum value. Furthermore, numeric and character string attributes may have their values limited to a specified range.

The following ASN.1 code indicates the possible structures of attribute values.

```
AttributeStructure ::= CHOICE {
                integer [0] IntegerConstraint,
                real [1] RealConstraint,
                characterstring [2] CharConstraint,
                name [3] NameConstraint,
                bitstring [4] NULL,
                octetstring [5] NULL,
                boolean [6] NULL,
                time [7] NULL }

IntegerConstraint ::= SEQUENCE {
                minimum INTEGER OPTIONAL,
                maximum INTEGER OPTIONAL,
                range SET OF INTEGER}

RealConstraint ::= SEQUENCE {
                minimum REAL OPTIONAL,
                maximum REAL OPTIONAL,
                range SET OF REAL}

CharConstraint ::= SET OF IA5String

NameConstraint ::= SET OF Name
```

The requirement for a flexible method of defining a wider range of attribute structures is discussed in section 8.6 of this thesis.

4.3.3.2. Examples of attribute definitions

The following examples illustrate some possible uses of attribute definitions.

The University of Nottingham might define an attribute called *title* for use within its entries. A person's title represents their position within the department and is specified by the definition below.

```
type = "title"
scope = /C=GB /O=Nott.Uni/
recurring = FALSE
structure = PrintableString
range = ("professor", "senior lecturer", "junior lecturer",
```

"research assistant", "technical staff", "secretary",
"postgraduate student", "undergraduate student")

This definition shows that a person may only have one title which is a printable string chosen from a limited range of values.

Another example is the following definition of a distribution list member.

type = "member"
scope = (root)
recurring = TRUE
structure = Name

This definition has global scope and allows an instance of a member attribute to have many values which are directory names.

4.3.3.3. Use of type and scope within attribute definitions

The previous sections have discussed the purpose and structure of attribute definitions and have introduced the concept of scope to limit the area of the DIT in which they are valid. The scope of attribute and entry definitions is important in supporting the management of information. This is fully discussed in section 4.3.5. However, the following paragraphs make some immediate observations concerning the relationship between attribute types and scopes.

- An attribute *type* may be ambiguous within the DIT. This means that a user might use the same type to refer to different classes of attribute within different contexts. For example, the attribute *role* might have different meanings within different classes of entry. This is a natural reflection of the way attributes are identified in the real world. The Directory is responsible for mapping from a user specified type to the correct definition.
- The combination of type and scope is unambiguous for an attribute definition. This means that there may not be two attribute definitions of the same type with exactly the same scope. However, there may be two attributes of the same type with nested scopes (i.e. nested subtrees). It follows that an attribute definition may be precisely identified by specifying its type and its scope.
- Although attributes corresponding to a given definition can only be created within its scope, they may be read by users who are not within this scope. For example, a user from the organisation */C=USA /O= MIT/* can read an attribute defined within the scope of the organisation */C=GB/ O=Nott.Uni/*. This may require retrieval of the attribute definition along with the attribute value to aid interpretation.

The conclusion of these points is that attribute types should be chosen by humans to reflect real world conventions. The choice of type is a critical one, but is not constrained to be an unambiguous one provided that the context (scope) of the definition is also specified.

The structure and purpose of attribute definitions have now been defined. The following sections discuss the effects of attribute definitions on the existing abstract operations as well as the introduction of new abstract operations to manage the definitions themselves.

4.3.3.4. The effect of attribute definitions on operations

Attribute definitions declare attributes within the Directory and specify constraints on the structure and contents of their values. These declarations and constraints must be observed during operations updating directory information. The abstract operations affected are *Modify Entry* and *Add Entry* which must check attribute definitions during the creation of a new attribute or the addition or replacement of an attribute value. Modifications are now performed subject to the following additional conditions.

- A new attribute may only be added if it is within the scope of an existing definition of the correct type.
- An attribute value may not be added if it would break a *recurring* constraint specified within the relevant attribute definition.
- A new or replacement attribute value must conform to the structure specified by the relevant attribute definition.
- A new or replacement attribute value must belong to any range and must fall between any maximum and minimum values specified by the relevant attribute definition.

The previous section noted that directory users outside of the scope of an attribute definition would need to retrieve the definition in order to interpret the relevant attribute values. This can be achieved by extending the *Read Entry* operation to return attribute definitions when indicated by a flag, set in its argument. The Results to Read Entry have now been extended to include both Access Control Lists and attribute definitions and this is shown by the following extension to the ASN.1 definition given in section 3.3.1

```
ReadResult ::= SET {
                [0] EntryInformation OPTIONAL,
                [1] SET OF ReadError }

EntryInformation ::= SET {
                entry [0] DistinguishedName,
                entrylevelacl [1] AccessControlList OPTIONAL,
                types [2] SET OF AttributeInformation }
```

```
AttributeInformation ::= SET {
                type [0] AttributeType,
                definition [1] AttributeDefinition OPTIONAL,
                attributelevelacl [2] AccessControlList OPTIONAL,
                values [3] SET OF AttributeValue }
```

4.3.3.5. New operations to manage attribute definitions

An important requirement of the directory integrity mechanism is that it should support the dynamic management of integrity controls. This requires extending the Directory Access Protocol to include operations allowing the creation and removal of attribute definitions. This section defines the *Add Attribute Def*, *Delete Attribute Def*, *Suspend Attribute Def* and *Reinstate Attribute Def* abstract operations for this purpose. In addition, it defines the *Read Attribute Def* operation allowing the general retrieval of attribute definitions. The introduction of new access categories to control these operations is also considered.

The *Add Attribute Def* operation adds the specified attribute definition to the Directory. It must ensure that there is no other definition with the same combination of type and scope. The ASN.1 specification of this operation is given below.

```
AddAttributeDef ::= ABSTRACT OPERATION
                ARGUMENT AddAttributeDefArgument
                RESULT AddAttributeDefResult
                ERRORS {NameError, AttributeDefError}

AddAttributeDefArgument ::= AttributeDefinition

AddAttributeDefResult ::= NULL
```

The *Delete Attribute Def* operation deletes an attribute definition identified by a combination of type and scope. Deletion can only occur if this definition is not "in use" within the Directory. The term "in use" implies that there should be no existing attribute instances corresponding to the definition and that it should not be referenced by an existing entry definition as described later. The ASN.1 specification of this operation is given below.

```
DeleteAttributeDef ::= ABSTRACT OPERATION
                ARGUMENT DeleteAttributeDefArgument
                RESULT DeleteAttributeDefResult
                ERRORS {NameError, AttributeDefError}

DeleteAttributeDefArgument ::= SET {
                type [0] PrintableString,
                scope [1] Name }

DeleteAttributeDefResult ::= NULL
```

Section 3.4 described the suspension of directory entries in order to support the phasing out of information. The same technique can be applied to definitions belonging to the directory integrity mechanism. An attribute definition may be marked as "suspended", therefore disallowing the creation of any new attributes of its class. Already existing attributes will then be deleted over a period of time after which the attribute definition itself can be deleted. It should also be possible to reinstate an attribute definition. These techniques require the introduction of the *Suspend Attribute Def* and *Reinstate Attribute Def* operations. Their arguments and results have similar structures to *Delete Attribute Def* and are fully specified in appendix A.

Finally, it should be possible to retrieve attribute definitions. This is supported by the extended Read Entry operation, and also by the new *Read Attribute Def* operation, returning the attribute definition with the specified type and scope. In addition, the type may be omitted, in which case this operation returns all definitions with the specified scope. The ASN.1 specification of this operation is given below.

```
ReadAttributeDef ::= ABSTRACT OPERATION
                ARGUMENT ReadAttributeDefArgument
                RESULT ReadAttributeDefResult
                ERRORS {NameError, AttributeDefError}

ReadAttributeDefArgument ::= SET {
                type [0] PrintableString OPTIONAL,
                scope [1] Name }

ReadAttributeDefResult ::= SET OF AttributeDefinition
```

The introduction of these operations requires the extension of the access control model to control their use. The retrieval and manipulation of attribute definitions is protected by the entry level ACL associated with their scope. Permission to perform the above operations requires the following access categories: *read_definition* allows an attribute definition to be retrieved. *add_definition* allows a new attribute definition to be created. *delete_definition* allows an attribute definition to be deleted, suspended and reinstated. The *Detect* right also applies to attribute definitions and allows them to be hidden if necessary.

Summary of attribute definitions

Attribute definitions specify the structure and contents of attributes. They may also place constraints on attribute values. Each attribute definition defines a specific attribute class within a specific scope. Scopes are represented by the subtrees of the DIT and define the space in which attributes of that class may exist.

The Add Entry and Modify Entry operations must ensure that the integrity constraints specified by attribute definitions are not broken. Finally, new abstract operations have been defined allowing the creation, suspension, deletion and retrieval of attribute definitions.

The following section uses attribute definitions as a basis for the *entry definition* defining the structures of entries and name-forms. The structure of entry definitions is described and further extensions to the Directory Access Protocol are specified allowing the dynamic management of entry definitions.

4.3.4. Entry definitions

This section describes *entry definitions* specifying the structures of entries and name-forms within the directory global conceptual model.

Section 3.1. mentioned that directory entries can be grouped into *object classes* where the entries of an object class are of the same type and have roughly the same structure. Examples of object classes might be *distribution list*, *person* and *device*. An entry definition declares the abstract structure of an object class and assigns it a type and a scope. It describes how entries are named and how they are structured in terms of mandatory or optional attributes. It may also specify default access controls and attribute values for entries belonging to an object class.

4.3.4.1. The structure of an entry definition

An entry definition is defined by the following ASN.1 structure.

```
EntryDefinition ::= SET {
        type [0] PrintableString,
        scope [1] DistinguishedName,
        contains [2] ObjectClass OPTIONAL,
        superiors [3] SET OF ObjectClass,
        rdntypes [4] SET OF AttributeTemplate,
        mandatory [5] SET OF AttributeTemplate OPTIONAL,
        optional [6] SET OF AttributeTemplate OPTIONAL,
        defaultacl [7] AccessControlList OPTIONAL,
        acl [8] AccessControlList OPTIONAL }
```

This relies on two further structures. An *object class* is a combination of type and scope unambiguously identifying an entry definition and hence a class of entries. An *attribute template* unambiguously identifies an attribute definition by a combination of type and scope and may also specify default access controls and values. These structures are defined below.

```
ObjectClass ::= SEQUENCE {
        type PrintableString,
```

```
                scope DistinguishedName }

AttributeTemplate ::= SEQUENCE {
                type PrintableString,
                scope DistinguishedName,
                defaultvalue AttributeValue OPTIONAL,
                defaultacl AccessControlList OPTIONAL }
```

The following paragraphs describe the role of each element in an entry definition.

The *type* is the handle by which users refer to the entry definition. It is also the label by which users refer to the object class of entries it defines. For example, *person*, *distribution list* and *organisation* are all types of entry definition and therefore classes of entry.

The *scope* defines the subtree of the DIT in which this entry definition is valid. This means that entries belonging to this object class can only exist within this scope. The scope of an entry definition plays the same role as that of an attribute definition and the combination of type and scope is therefore unique.

The *superiors* element defines the object classes which may be immediately superior to this class in the DIT. The *rdntypes* element lists the possible attribute classes used in forming the relative distinguished name of an entry of this class. The recursive combination of these two elements determines possible structures of distinguished names for entries of this class. Thus, entry definitions are used to define possible directory name-forms in terms of the structures of relative distinguished names and the hierarchical relationships between object classes (i.e. a name form is specified by a superior name form and a naming attribute).

It is important to note that, whenever an entry definition refers to an attribute, it specifies both its type and scope in order to unambiguously identify the relevant attribute definition.

The *mandatory* element describes those attributes which must be present in all entries of this class. It also suggests default ACLs and values for these attributes.

The *optional* element describes all other attributes which may be present in entries of this class along with default ACLs and values.

The *defaultacl* suggests a default entry level ACL for entries of this class.

The *acl* is the access control list controlling access to the entry definition itself.

Finally, the *contains* element may identify an entry definition contained within this one. This facility allows local refinement of more global definitions. It is illegal for an entry definition and a contained definition to conflict with each other and any number of levels of nesting are allowed. Containment requires that the following conditions are applied recursively.

- The scope of the new definition is the same or within the scope of the contained definition.
- Attributes described in an entry definition may not have overlapping types with those in its contained definition.
- Suggested Access Control Lists in an entry definition override those in its contained definition.

In summary, an entry definition is identified by a combination of type and scope in the same way as an attribute definition. Entry definitions describe the structure of entries by specifying mandatory and optional attributes including default access controls and values. They specify directory name-forms in terms of the attributes forming relative distinguished names and the permitted parent-child relationships between different object classes. Entry definitions may also be nested.

The following section describes the effect of entry definitions on the abstract operations used to update information. This is followed by the specification of new abstract operations to manage entry definitions themselves.

Examples of entry definitions are given in section 4.3.7

4.3.4.2. The effect of entry definitions on operations

Entry definitions declare and constrain the structure of entries and names within the Directory. These constraints must be observed during those operations manipulating the Directory Information Tree and the contents of entries. The operations affected by these constraints are *Add Entry* and *Modify Entry*.

Each entry is added to the Directory by an Add Entry operation and must conform to a valid entry definition at the time of its creation. This implies the following constraints for the Add Entry operation.

- The new entry must have an object class identifying an existing entry definition.
- The new entry must be within the scope of the relevant entry definition.
- The proposed parent of the new entry must have an object class listed in the *superiors* element of its entry definition.
- The relative distinguished name of the new entry must be composed of attributes from the *rdntypes* element of its entry definition.
- All mandatory attributes must be present in the new entry.
- Any other attributes in the new entry must be optional within its definition.

In addition, the operation can utilise any default attribute values and ACLs specified by the entry definition.

Once an entry has been added to the Directory, it must continue to obey the definition of its object class. This has the following implications for the Modify Entry operation.

- A mandatory attribute cannot be deleted.
- An attribute cannot be added which is not mandatory or optional.

All of the above conditions apply recursively with the constraints specified in nested definitions.

4.3.4.3. New operations to manage entry definitions

This section extends the Directory Access Protocol to include abstract operations for creating, deleting, suspending and reading entry definitions in order to support the dynamic management of directory integrity constraints. These operations are similar to those reading and manipulating attribute definitions and, consequently, this section only describes their major features. The abstract operations concerned with entry definitions are listed in the table below and briefly described in the following paragraphs.

operations for entry definitions
Add Entry Def
Delete Entry Def
Read Entry Def
Suspend Entry Def
Reinstate Entry Def

The *Add Entry Def* operation adds the specified entry definition to the Directory. In order to successfully add a new entry definition, the following conditions must be satisfied.

- There should be no other entry definition with the same type and scope.
- The object classes listed as possible superiors should be valid and the new definition must fall within their scopes.
- Any immediately contained entry definition must exist and the new definition must be within its scope.
- This definition must not conflict with any contained definition.
- All referenced attribute definitions should be valid and the new entry definition must be within their scopes.

The *Delete Entry Def* operation is used to remove entry definitions from the Directory. Before deleting an entry definition it must ensure that no entries exist belonging to this object class and that no other entry definitions contain this object class or specify it as a possible superior.

The phasing out of entry definitions is also supported by use of the *Suspend Entry Def* operation which marks them as "suspended" and disallows them from being used in the creation of new entries or definitions. Suspended entry definitions may be reinstated by the *Reinstate Entry Def* operation. Suspension may also give the effect of modification for definitions. A definition may be suspended and a new definition may be created with the same type and scope to replace it. This enables effective modification of a definition without causing conflict with existing information.

Finally, it is possible to retrieve entry definitions by the *Read Entry Def* operation taking a type and scope as its argument. The type may be omitted allowing the retrieval of all entry definitions with a given scope.

These operations are controlled by the directory access control mechanism using the same access categories as for attribute definitions. In order to add a new entry definition it is necessary to possess the *add_definition* access right at the entry representing its scope. Deletion and suspension require the *delete_definition* access category within the entry definition's ACL and retrieval requires the *read_definition* category, also within this ACL.

Summary of entry definitions

Entry definitions define the structure of entries and name-forms within the Directory. The structure of entries is specified in terms of mandatory or optional attributes and default values and ACLs. Permitted name-forms are determined by the attributes allowed within relative distinguished names and constraints on hierarchical relationships between object classes in the DIT.

Each entry definition is valid within a specified subtree of the DIT called its scope. An entry definition may also contain other recursively nested definitions allowing the local modification of more global integrity constraints.

This section has outlined the effect of entry definitions on those abstract operations updating directory information. It has also extended the Directory Access Protocol to include new operations to read and manipulate the entry definitions themselves.

The next section discusses the issue of *scope* in greater detail and explains how it supports the concept of Information Management Domains.

4.3.5. Scope and Information Management Domains

Scope is a crucial aspect of the directory integrity mechanism. This section describes the role of scope in representing the policies of different Information Management Domains and explains how the Directory maps possibly ambiguous attribute types to unique entry and attribute definitions.

The management of directory information requires that different Information Management Domains can define their own integrity policies. Scope limits the effects of definitions to DIT subtrees representing natural IMD boundaries based on organisational hierarchy. Each IMD can therefore define its own integrity constraints in terms of entry and attribute definitions with local scope. The use of scope in this way has several interesting consequences.

- Separate IMDs can support different definitions with the same types. For example, two IMDs might choose to define the structure of *distribution list* entries differently.
- There may be some inheritance of definitions and hence policy between hierarchical IMDs. For example, the root of the DIT can be seen as representing an IMD defining global policies which might be inherited, refined or redefined by other IMDs.
- The refinement of policies between hierarchical IMDs is supported by the nesting of *contained* entry and attribute definitions.

The use of scope to represent external IMDs within the DIT is shown by figure 4.8.

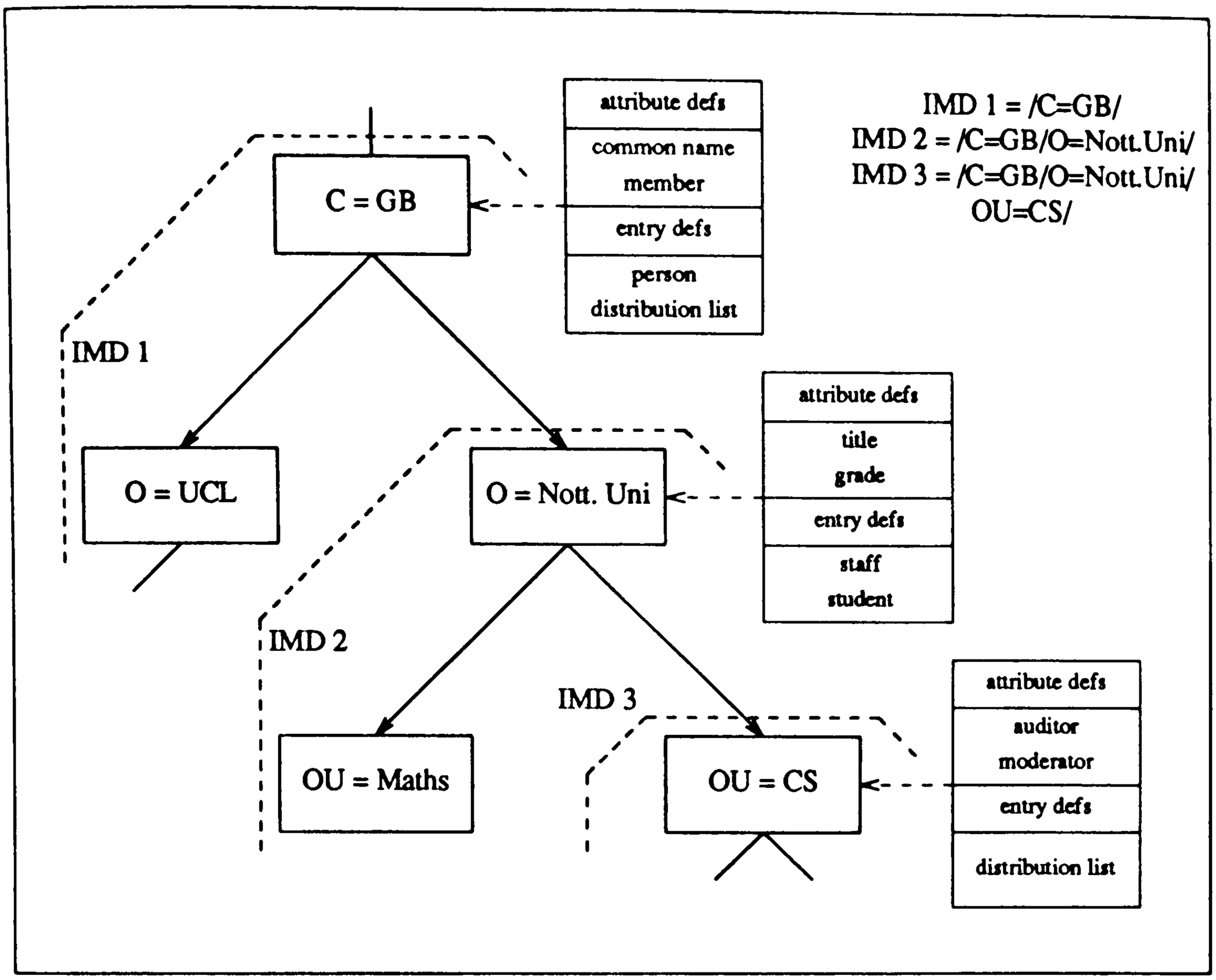


Figure 4.8 Scope representing IMDs within the DIT

The figure shows a number of IMDs (dashed lines) superimposed on a portion of the Directory Information Tree. Each of the three IMDs shown defines local integrity policies via attribute and entry definitions whose scope is the root of the IMD subtree. In general, the policies of an IMD are inherited by all inferior IMDs, although they may also be redefined. For example, IMD 3 inherits the "person" entry definition and redefines the "distribution list" entry definition from IMD 1.

These points have shown the use of scope in reflecting the boundaries of Information Management Domains. Without the ability to limit the scopes of definitions, the integrity mechanism would soon become unmanageable and, consequently, so would directory information. Scope is therefore fundamental to the management of integrity constraints. It should be noted that X.500 does not support the dynamic management of its integrity rules (schemas) and does not allow the scoping of schemas which would make this possible. This issue is explored further in section 8.5.

The introduction of scope means that *types* may be ambiguous. Most users will refer to attributes and object classes by type alone and it is therefore necessary to specify how the Directory resolves ambiguous types to unique definitions. Resolution works in the following way.

- Whenever a user manipulates an existing attribute or entry the Directory can map the supplied entry name to its entry definition which can then be inspected to obtain the types and scopes of relevant attribute definitions.
- Whenever the user creates a new entry or entry definition they have the choice of identifying any relevant definitions by supplying both type and scope themselves or by supplying the type alone and allowing the Directory to resolve to the definition with the closest scope (i.e. the most local definition).

Resolving to the definition with closest scope

The closest scope is the most local, valid scope from the user's point of reference. The following example shows how the Directory uses the closest scope rule to resolve ambiguous types.

Consider two entry definitions having the type *person* with scopes */C=GB/* and */C=GB /O=Nott.Uni /OU=CS/* respectively. If a user wished to create the entry */C=GB /O=Nott.Uni /OU=CS /CN=Steve/* of type person, the closest scope rule would resolve to the second definition, whereas, if they wished to create the entry */C=GB /CN=Steve/* of type person, it would resolve to the first definition.

In conclusion, a user may refer to definitions by type alone, in which case the Directory finds the most local definition. Alternatively, the user may override this rule by supplying both the type and scope of a definition themselves.

4.3.6. Dynamic management and attribute types

A major goal of the data integrity mechanism is support for dynamic management. This is provided by new abstract operations to read and manipulate entry and attribute definitions. These operations, in turn, are supported by the inclusion of human readable attribute types within the directory information model.

The dynamic definition of new attribute types implies that user interfaces must cope gracefully with previously unseen attribute types (for example, a newly defined attribute type in a remote part of the DIT). This is a particular problem for human interfaces where new types must be presented in a human readable form. This problem is eased by the inclusion of human readable attribute types within attribute definitions so that they may be retrieved by

the user interface and examined for a suggested attribute type.

It should be noted that the problem is not entirely solved by this mechanism because the user may not understand the semantics behind the suggested label. This is particularly true when crossing language barriers. However, in many cases, users will stand a good chance of understanding the new type as it is presented to them.

The above solution is in direct contrast to the X.500 approach where the mapping to human readable types is performed within the user interface. The X.500 approach is generally unable to support the dynamic definition of new attributes. Furthermore, it requires the implementation of this mapping within each interface. Consequently, altering the mapping by defining new attribute classes is extremely expensive in terms of effort.

4.3.7. Summary and examples of the directory integrity mechanism

This section concludes the description of the directory integrity mechanism by presenting a brief summary of attribute and entry definitions and giving some examples of their use.

The directory integrity mechanism facilitates the correct update of directory information by the provision of attribute definitions and entry definitions specifying the structure of attributes and entries respectively. Entry definitions also specify possible directory name-forms. Definitions can be seen as templates for directory information and can therefore be used by DUAs to support user friendly interfaces based on a *Query by Example* mechanism.

The previous sections have considered the following issues.

- Specifying the structure of entry and attribute definitions.
- Describing the effect of entry and attribute definitions on existing abstract operations.
- Specifying new abstract operations to support the dynamic management of entry and attribute definitions.
- Specifying a scope mechanism for entry and attribute definitions which can be used to support the concept of Information Management Domains.

These last two points are of major importance to this work.

The following are examples of the use of the data integrity mechanism. The examples occur within two Information Management Domains. The first is an IMD for the University of Nottingham represented by the subtree with root */C=GB /O=Nott.Uni/*. The second is a sub domain for the department of Computer Science within the University of Nottingham represented by the subtree with root */C=GB /O=Nott.Uni /OU=CS/*

Example 1: defining a distribution list

Consider the case where the University of Nottingham runs a campus wide mail service supporting its own distribution list protocol. Each distribution list names a group of submitting members, a group of receiving members and a single list moderator. In addition, each list has a human readable text name. The mail service uses the Directory service to store the descriptions of distribution lists. This requires the definition of a distribution list object class.

The properties of a distribution list can be represented by the following attribute definitions.

Attribute definitions (scope: /C=GB/O=Nott.Uni/)		
type	structure	recurring
list name	printable string	no
submitting member	name	yes
receiving member	name	yes
moderator	name	no

A distribution list entry is defined by the following entry definition. This assumes the existence of the global entry definitions for *organisation* and *org unit* and refers to other definitions by type with scope in parentheses.

Entry definition	
type	distribution list
scope	/C=GB/O=Nott.Uni/
superiors	organisation (root) org unit (root)
rdntypes	list name (/C=GB/O=Nott.Uni/)
mandatory	submitting member (/C=GB/O=Nott.Uni/) receiving member (/C=GB/O=Nott.Uni/) moderator (/C=GB/O=Nott.Uni/)

This allows a distribution list entry to be immediately inferior to an *organisation* or *organisational unit* entry in the DIT and ensures that the list name is used for its RDN. Thus the name of a distribution list always has the form: */.../O= /list name= /* or */.../OU= /list name= /*. This example also shows that a distribution list can be guaranteed to have exactly

one moderator by making this a mandatory, non-recurring attribute in its definition.

Example 2: redefining a distribution list

Suppose now that the Computer Science department at Nottingham was developing a new experimental mail service allowing distribution lists to optionally include some additional properties such as a text description, list administrators and reply-to users. The new distribution lists also included a mandatory auditor to deal with delivery reports arising from use of the service. Ideally, the implementors would like to test the new service and run the old campus wide service simultaneously.

This could be achieved by firstly defining the following new attribute types.

Attribute definitions (scope: /C=GB/O=Nott.Uni/OU=CS/)		
type	structure	recurring
description	printable string	no
reply to	name	yes
administrator	name	yes
auditor	name	no

The definition of a distribution list could then be extended by the following new entry definition.

Entry definition	
type	distribution list
scope	/C=GB/O=Nott.Uni/OU=CS/
superiors	org unit (root)
rdntypes	list name (/C=GB/O=Nott.Uni/)
contains	distribution list (/C=GB/O=Nott.Uni/)
mandatory	auditor (/C=GB/O=Nott.Uni/OU=CS/)
optional	administrator (/C=GB/O=Nott.Uni/OU=CS/) description (/C=GB/O=Nott.Uni/OU=CS/) reply to (/C=GB/O=Nott.Uni/OU=CS/)

This example shows how definitions can be nested. The advantage of nesting the new definition inside the old is that the new distribution lists would be guaranteed to contain the correct information necessary to run under the old as well as the new service.

The Directory would now contain two definitions of type *distribution list* but with different scopes. We can use this scenario to show the use of the *closest scope* rule in resolving types to definitions. If a user were to create the distribution list entry */C=GB /O=Nott.Uni /OU=CS /list name=CS social/* the Directory would use the new definition. If they were to create a similar entry named */C=GB /O=Nott.Uni /list name=Cricket Club/* the Directory would use the old definition.

In the first case, the user could force usage of the old definition by supplying a scope and type for the object class being created.

One can imagine how a DUA might provide a friendly interface for the creation of a distribution list by downloading the relevant entry definition and prompting the user for the mandatory and optional attributes.

These examples conclude the discussion of the directory integrity mechanism. The final section of chapter 4 summarises information management and the global conceptual model.

4.4. Review of the directory global conceptual model

Chapters 3 and 4 of this thesis have specified the directory global conceptual model. The following briefly summarises this work and emphasises its important features.

The role of the global conceptual model is to represent the information content of the Directory in an abstract form, independent of specific user applications, distribution issues and underlying storage systems. The global conceptual model is also responsible for describing the access controls and integrity constraints applicable to directory information.

The information model and abstract operations

Chapter 3 described the basic directory information model and abstract operations. In this model, communication entities are represented by *entries* containing sets of *attributes*. Entries are arranged into a hierarchical structure called the *Directory Information Tree* responsible for naming them. Each entry has a unique *distinguished name* but may also be identified by a number of alternative names called *aliases*.

Users' actions on the Directory are described in terms of abstract operations providing the functionality to read, browse, modify, create and delete information in a general way. A mechanism for phasing out information was specified by the *Suspend Entry* and *Reinstate Entry* operations.

Information management

Chapter 4 extended the basic model of chapter 3 by considering the problem of information management. The meaning of information management was considered and a model was developed dividing the Directory into *Information Management Domains* reflecting organisational policies. The remainder of chapter 4 was devoted to specifying global directory access control and integrity mechanisms to support the management policies of different IMDs.

The access control mechanism is based on Access Control Lists associated with individual entries and attributes. The important aspects of this work are the *access category* and *Object Set Descriptor* mechanisms for describing actions and groups of users in a flexible and dynamic way.

The integrity mechanism allows the specification of directory entries, attributes and nameforms by two structures called *entry definitions* and *attribute definitions*. Two important aspects of the integrity mechanism are the ability to manage definitions dynamically using abstract operations and the concept of *scope* supporting Information Management Domains.

Major differences to X.500

This research has occurred in parallel with the development of X.500 and like X.500, has used earlier work on nameservers such as *The Clearinghouse* as a base. However, this thesis has adopted a more database oriented view of the Directory service and has produced different results, particularly in the area of information management. The following are the major differences between the global conceptual model of chapters 3 and 4 and the X.500 *Information Framework* and *Service Definition*.

- This research allows the addition and deletion of non-leaf DIT entries.
- This work supports the phasing out of information via the *Suspend* operation and the resetting of aliases.
- This work supports an integrated access control mechanism and allows the management of access controls via abstract operations.
- This work supports an integrity mechanism allowing the dynamic management of integrity constraints via abstract operations.

- This work actively supports Information Management Domains by the concept of scope and containment of definitions.

The last three points are of the most importance. Following its removal from the Melbourne output, X.500 has left the specification of an access control mechanism to the next study period and, although it supports *schemas* for integrity control, at the present they have to be defined by an extra-directory mechanism. As a result 1988 X.500 cannot effectively support information management on a global scale.

The emphasis on standardisation within X.500 means that many attribute types and object classes have been defined as part of the standard. This is not a flexible way of supporting information management and does not easily adapt to different policies. Without flexible and dynamic information management tools, directory information will be difficult to manage.

This concludes the specification and discussion of the directory global conceptual model. Chapters 5 and 6 turn their attention to the local conceptual layer and consider how the above functionality can be realised in a distributed system. This involves discussion of partitioning, navigation, distributed operations and replication.

Chapter 5

The Local Conceptual Layer: Knowledge, Navigation and Operations

This chapter is concerned with the development of the directory local conceptual model.

Previous chapters have considered the Directory as if it were a centralised database system. However, the Directory will in fact be a distributed system. This chapter therefore considers how the global conceptual model may be distributed between a number of database nodes called *Directory System Agents* (DSAs). This work includes partitioning the Directory Information Tree between DSAs, describing an operation navigation procedure and examining the distributed execution of operations.

Once again, management issues form a major part of this work. These are primarily concerned with the management of the distributed Directory system itself and, in particular, with the management support required for system reconfiguration as organisations and networks change. In addition, this chapter also considers the support required for the distributed operation of the directory access control and data integrity mechanisms specified in chapter 4.

The following section provides an introduction to the distributed directory model and identifies the major issues to be addressed by this work.

5.1. An overview of distribution

Distribution of the Directory service concerns the realisation of directory functionality within a system of autonomous DSAs connected by communication links. Distribution is necessary for the following reasons.

- The Directory will be an extremely large database accessed on a global scale. A centralised Directory would be very expensive in terms of communication costs.
- The Directory will be provided by many cooperating organisations. It will not be politically acceptable for any single organisation to store and maintain all directory information.

- Distribution reduces the bottleneck and robustness problems occurring within large scale centralised systems. Centralised nameservers such as the *CSNET Nameserver* [LAN83] and *JNT Name Registration Scheme* [LAR82a] are unlikely to scale to global proportions due to these problems.

These factors clearly indicate the requirement for the Directory to be a distributed service. This section provides an overview of the major issues to be considered when specifying the distribution of the Directory.

- Section 5.1.1 reviews the X.500 directory functional model adopted as the basis for all later work.
- Section 5.1.2 discusses general distribution issues.
- Section 5.1.3 outlines the specific distributed management issues to be addressed by this work.

5.1.1. Functional model and general distributed operation

Figure 5.1 shows the elements of the directory functional model as specified within X.500.

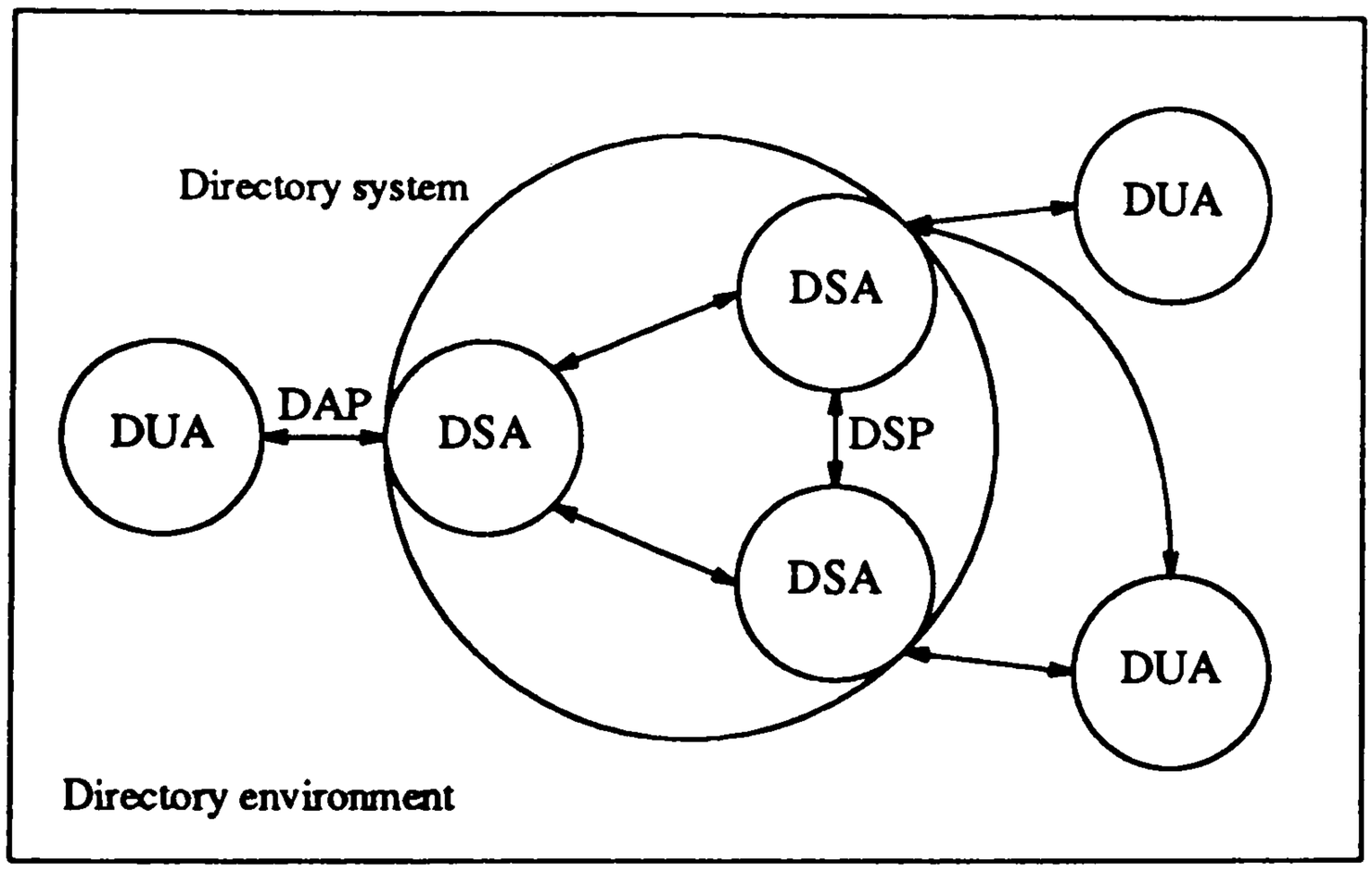


Figure 5.1: Elements of the directory functional model

Directory operations are performed by physically distributed server entities called *Directory System Agents* (DSAs). The Directory Information Tree is partitioned among the set of DSAs so that each DSA is *responsible* for a subset of the entries in the DIT. Responsibility implies that the DSA is able to answer queries concerning these entries.

The Directory performs operations in the following way: A user connects to the Directory via their *Directory User Agent* (DUA) which establishes contact with a DSA. The user's DUA then requests operations at this DSA. If the DSA is responsible for the entries specified in the operations, it will execute them and return the results. Otherwise, the operation will be navigated to a responsible DSA and executed there. The results will then be returned to the user. The responsible DSA may interact with other DSAs to execute the operation.

The navigation of directory operations may involve a number of DSA interactions before a responsible DSA is located. These should be transparent to the user thus preserving location transparency. Furthermore, navigation is distributed, meaning that each DSA involved in navigation acts autonomously and that the information required for navigation is distributed among DSAs.

The directory supports several modes of operation.

- *Chaining* occurs when each DSA navigating an operation recursively calls the requested operation at the next DSA.
- *DSA referral* occurs when one DSA assumes overall responsibility for navigation and contacts a sequence of other DSAs which return *referrals* eventually leading to a responsible DSA.
- *DUA referral* is similar to DSA referral, except that it is the user's DUA assuming overall control of navigation.
- Some operations may require *multicasting* and *query decomposition* where an operation is split into several sub-operations which are then navigated to different DSAs for execution.

These modes of operation were shown in figure 1.13 and their relative merits are discussed in section 5.5. DUA/DSA interactions are specified by the abstract operations of chapters 3 and 4 forming the *Directory Access Protocol* (DAP). DSA/DSA interactions are specified by the *Directory System Protocol* (DSP) as described later.

This concludes the description of the directory functional model. The following sections outline the major issues to be addressed when using this model to specify the distributed operation of the Directory service.

5.1.2. Major distribution issues

Specifying the distributed operation of the Directory requires mapping from the global conceptual layer to the local conceptual layer of the directory architecture. There are a number of issues relevant to this mapping.

- The Directory Information Base must be partitioned between a number of DSAs. This involves the systematic division of the DIT between DSAs and the introduction of *knowledge* information describing the responsibilities of DSAs.
- A navigation procedure must be specified allowing information to be located within the distributed system and operations to be directed at suitable DSAs.
- The sequence of DSA interactions required to execute each operation should be examined and new DSP operations should be specified to support these interactions. This task corresponds to analysing *query processing* [DATE83] within general database systems.
- A replication mechanism is required to support the consistent maintenance of *replicated* information between DSAs.

These issues form the basis of the following work. In addition to these general issues, distribution of the Directory introduces a number of management issues. These are described by the following section.

5.1.3. Management issues within the local conceptual layer

This thesis is particularly concerned with directory management issues. There are a number of management issues relevant to the local conceptual layer.

Firstly, the Directory should support the management of information as described in chapter 4. This requires specifying support for the distributed operation of the directory access control and integrity mechanisms.

Secondly, mechanisms should be provided to support the management of the distributed Directory service itself. Both X.500 and existing nameservers have considered the issues of navigation and distributed operations. However, these services generally fail to address the problems of adapting to changes in the organisations, agents and underlying networks providing the services. These problems are highlighted by the following questions.

- How can the Directory guarantee connectivity between DSAs when parts of the system are reconfigured?
- How are new configurations propagated to other DSAs and incorporated in their distribution knowledge?
- How can the Directory use distribution knowledge to optimise navigation under different circumstances?
- How can the Directory operate with (and correct) inconsistent distribution knowledge acquired as a result of caching and similar methods?

In general, mechanisms are required to allow the Directory system to react gracefully to reconfiguration.

A third management issue concerns the management of replication. Mechanisms should be specified supporting the establishment and maintenance of replication agreements between DSAs and the consistent update of replicated information.

These management issues form an important part of this chapter and also of chapter 6. The remainder of this chapter is structured in the following way.

- Section 5.2 specifies the partitioning of the Directory Information Tree between DSAs and describes distribution knowledge and navigation. It pays particular attention to specifying the basic knowledge required to guarantee successful navigation as well as the knowledge used to optimise navigation and increase the robustness of the Directory.
- Section 5.3 discusses the management of distribution knowledge in the face of system reconfiguration. It explains how knowledge inconsistencies can be automatically detected and corrected by the Directory system.
- Section 5.4 describes the distributed operation of the Directory system and the interactions between DSAs required to execute operations. It also specifies new operations belonging to the Directory System Protocol.
- Section 5.5 presents a summary of the work in this chapter.

Replication is the subject of chapter 6.

5.2. Partitioning the DIT, knowledge and navigation

This section describes the partitioning of the Directory Information Tree between DSAs and discusses the navigation of operations within the Directory system. It also specifies the *knowledge* information required by each DSA in order to perform navigation.

Section 5.2.1 defines navigation. Section 5.2.2 specifies how the DIT is partitioned between DSAs and how this is represented by knowledge information. Section 5.2.3 then defines *minimal knowledge*, used in specifying a basic navigation algorithm in section 5.2.4. This is extended to include optimisation using *opportune knowledge* in section 5.2.5.

5.2.1. Navigation

Navigation is the process of directing an operation to a DSA able to perform it.

Each user requested operation will enter the Directory at an initial DSA and will be navigated via some intermediate DSAs to a responsible DSA.

The navigation of operations within the Directory service requires that it is always possible to determine the responsible DSA for a target entry from any initial DSA a user might contact. The process of locating the responsible DSA given the name of an entry is the fundamental goal of navigation and may require interaction between a sequence of DSAs called a *navigation sequence*. To be precise, navigation may be defined as the process of mapping a directory name onto the name and address of the DSA responsible for the named entry.

Each DSA in the navigation sequence performs a *navigation step*. This is a mapping at a single DSA between a directory name and the name of a DSA *closer* to the responsible DSA for the named entry. The term *closer* is explained later. The navigation process is shown in figure 5.2.

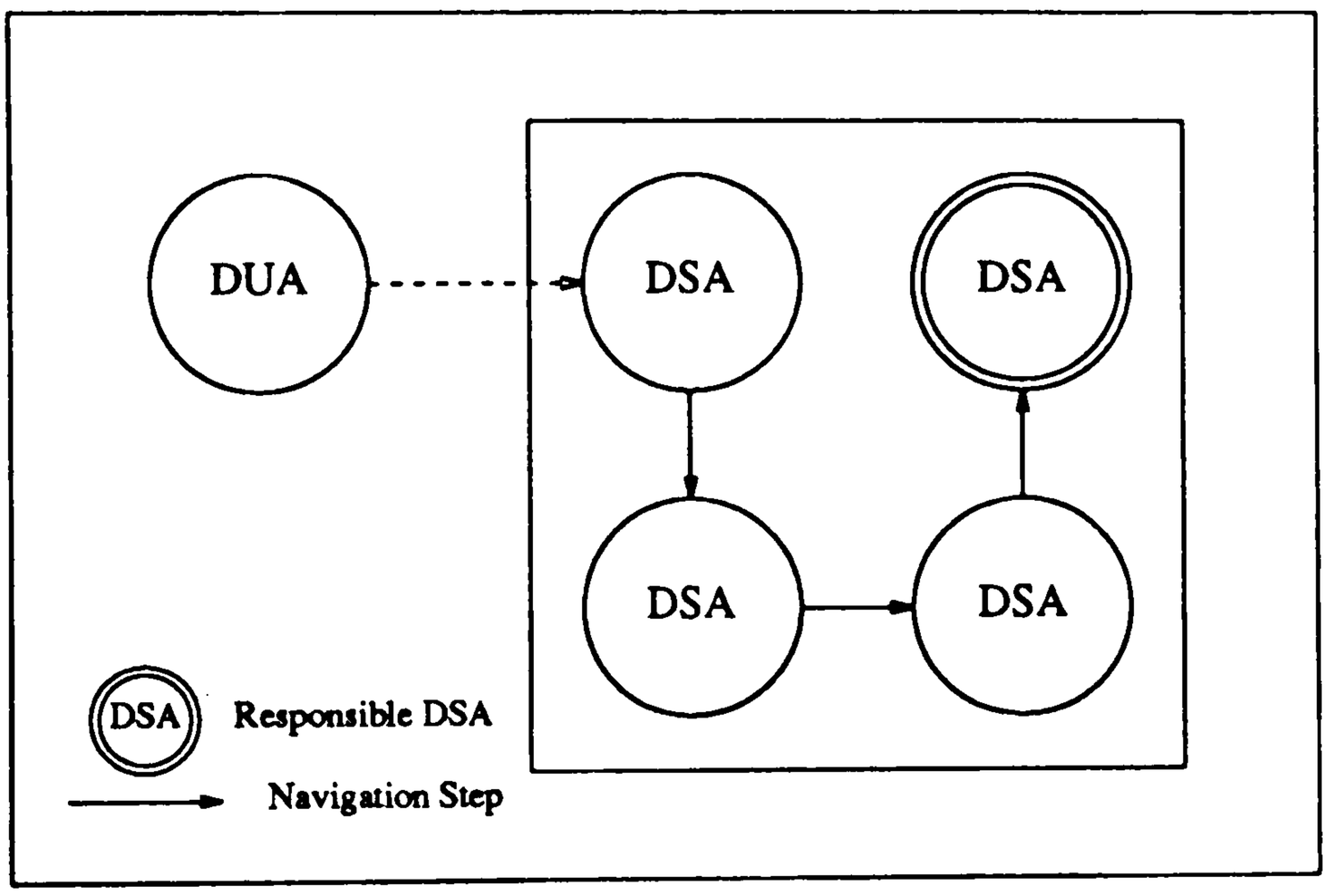


Figure 5.2 Navigation as a sequence of navigation steps

Directory navigation is decentralised, meaning that each DSA is able to perform a navigation step for any possible name independently of other DSAs. This requires that each DSA is aware of the responsibilities of some other DSAs in order to determine a closer DSA to the target entry. The information describing the responsibilities of other DSAs is called *knowledge*. Knowledge is vital to navigation and, therefore, to the distributed operation of the Directory service. Consequently, its structure is discussed in the following section which specifies the partitioning of the DIT between DSAs.

5.2.2. Partitioning the Directory Information Tree

The following paragraphs describe the division of the DIT into fragments, distributed between a set of DSAs. Fragments will then be used as the basis for the *knowledge* used in navigation.

Fragments

The Directory Information Tree (DIT) can be divided among a set of DSAs where each DSA is responsible for a set of entries grouped into *fragments*. The fragment is the basic unit of distribution and takes the form of an incomplete subtree of the DIT, extending from an entry down to a set of leaf or non-leaf entries. The *fragment name* is the name of the root entry of this incomplete subtree. Although fragmentation could occur at the level of individual entries, it will most likely occur on an organisational basis closely following the structure of the DIT. For example, the DIT subtree representing the University of Nottingham might be a fragment stored within a single DSA.

An example mapping between the DIT, DSAs and fragments is shown in figure 5.3.

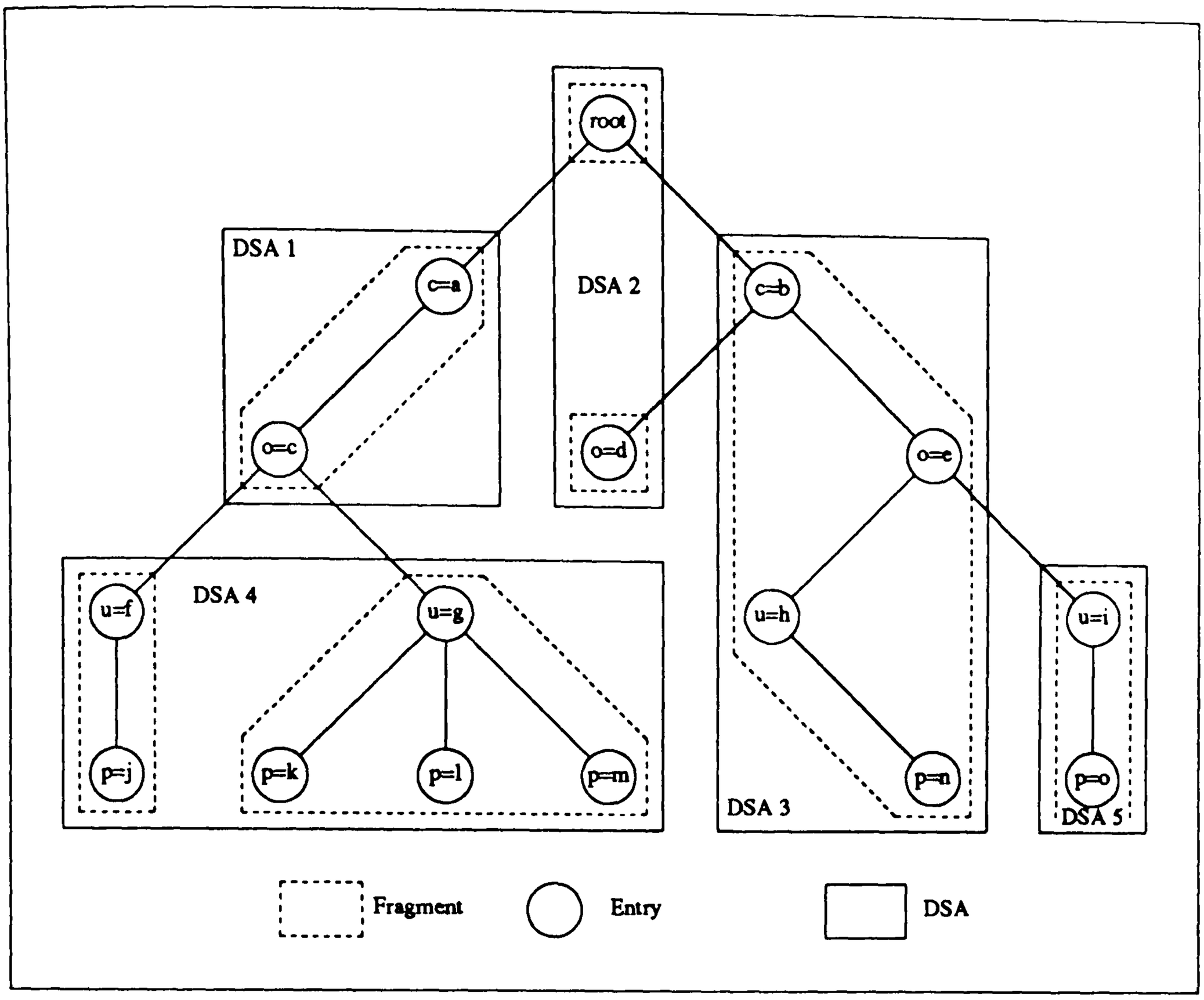


Figure 5.3: The DIT partitioned into fragments distributed between DSAs.

Figure 5.3 indicates that a DSA may be responsible for more than one fragment. Furthermore, each fragment belongs to exactly one DSA and, therefore, there is no replication of entries among DSAs. Chapter 6 removes this restriction and extends the model to include replication.

Knowledge and references

Navigation requires that each DSA is aware of the responsibilities of other DSAs. These responsibilities can be expressed in terms of fragments and this awareness is the DSA's *knowledge* as described above. Knowledge can therefore be modelled as a set of *references* describing the fragments for which DSAs are responsible. A reference associates the name and address of a DSA with the names of the fragments for which it is responsible. The following ASN.1 structure defines a reference:

```
Reference ::= SEQUENCE {
                dsa DistinguishedName,
                address, -- application entity address
                responsibilities SET OF DistinguishedName }
```

The *dsa* element specifies the distinguished name of a DSA and the *responsibilities* element lists the names of the fragments for which this DSA is responsible.

For example, the reference describing the responsibilities of DSA 4 in figure 5.3 is:

```
{ dsa: name of DSA 4,
  address of DSA 4,
  responsibilities: "/c=a/o=c/u=f", "/c=a/o=c/u=g" }
```

This definition of a *reference* requires that DSAs are explicitly named within the Directory. The use of distinguished names for DSAs serves the following purposes:

- The separation of DSA names from addresses allows the DIT to be partitioned among a logical set of DSAs independently of their physical locations.
- DSAs may need to be represented within the Directory for authentication purposes.

The choice of DSA names may be determined by the administrators of the *Directory Management Domains* [CCITT-X500] supporting them. In addition, DSA naming constraints may be imposed by a particular Directory implementation. For example, the QUIPU Directory service adopts the convention that:

> "DSAs should be named after endangered South American Wildlife" [KIL88b].

In general, the important issue of DSA naming is part of the general problem of naming application processes and entities. This is being addressed as part of the extended OSI reference model (*naming and addressing*) [ISO-OSI-3].

A DSA's knowledge can be divided into three categories: *minimal*, *opportune* and *replicated* where each category is distinguished by the manner in which it is acquired and managed.

- A DSA's minimal knowledge is the subset of its references essential for the DSA to perform a navigation step. Minimal knowledge is acquired and managed by a well defined protocol.
- A DSA's opportune knowledge is a set of references used in optimising the navigation process. Opportune knowledge is typically acquired and managed by ad-hoc methods such as caching.
- A DSA's replicated knowledge is a set of references also used for optimising the navigation process. Replicated knowledge is acquired and managed during the general

replication of information.

Each of the above categories exhibits a different degree of reliability and is utilised accordingly. Minimal knowledge is examined within the next section and is used to develop a basic navigation step algorithm. Opportune knowledge is specified in section 5.2.5 and is used to extend the navigation algorithm. Replicated knowledge is discussed in chapter 6.

5.2.3. Minimal knowledge

Minimal knowledge specifies the fundamental connectivity between all DSAs and is required to ensure that a DSA can always perform a correct navigation step for any target entry specified in an operation. This is guaranteed if the DSA is always able to determine one of the following:

- whether the DSA is responsible for the target entry.
- the name of another DSA *closer* to the target entry.

There are several ways in which this could be achieved. For example, each DSA could hold the reference of every other DSA in the Directory system. However, this would require much coordination and updating of knowledge as DSAs were added and removed. Instead, the proposed definition of minimal knowledge will specify the minimum number of references maintained at each DSA to guarantee a correct navigation step and hence overall navigation.

Before specifying minimal knowledge, it is first necessary to define some terminology: Let DSA A be responsible for fragment(s) A and DSA B be responsible for fragment(s) B

- If the fragment name of B is in the subtree of fragment A then DSA B is an *inferior DSA* of DSA A.
- If a fragment name of B is a direct child of fragment A then DSA B is an *immediately inferior DSA* of DSA A.
- If DSA B is an *inferior DSA* of DSA A then DSA A is a *superior DSA* of DSA B.
- If DSA B is an *immediately inferior DSA* of DSA A then DSA A is an *immediately superior DSA* of DSA B.
- DSAs A and B are *adjacent* if they form an *immediately inferior/superior* pair.

For example, referring to figure 5.3, DSA 3 is an *immediately inferior DSA* of DSA 2, but DSA 5 is only an *inferior DSA* of DSA 2. DSAs 2 and 3, as well as DSAs 3 and 5, are *adjacent*. It should be noted that, because each DSA may be responsible for many fragments, it may have many immediately superior and inferior DSAs.

The structure of minimal knowledge specified by this section ensures that a DSA can either determine whether a target entry is local or can navigate the operation to an adjacent DSA closer to the target. Thus, the process of navigation moves the operation between adjacent DSAs until the target is reached. For example, an operation initialised at DSA 5 with the target entry */c=a/o=c/u=f/* would be navigated via the ordered sequence DSA 5, DSA 3, DSA 2, DSA 1 and finally DSA 4.

It follows that a DSA can perform a correct navigation step if, for any target entry, it implements the following steps:

1. Determine whether the target entry would belong to a fragment which is the responsibility of this DSA.
2. Determine whether the target entry would belong to a fragment which is the responsibility of an inferior DSA.
3. Determine the name of an immediately superior DSA if none of the above are true.

In order to perform these steps, the minimal knowledge of the DSA must contain references for itself, all immediately superior and immediately inferior DSAs. Consequently, all DSAs providing the Directory service are connected in their own graph structure. (Not a tree because each DSA may have many immediate superiors!)

The structure of a DSA's minimal knowledge is defined by the following ASN.1:

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

This structure describes the minimal knowledge required for all of the fragments belonging to a DSA. For each of these fragments, the DSA sees a reduced tree structure called a *knowledge tree* containing the entries in the fragment and references to those DSAs adjacent to the fragment. If we consider DSA 3 from figure 5.3, we see that its minimal knowledge must contain references to itself and DSAs 2 and 5. The knowledge tree for DSA 3 is shown in figure 5.4.

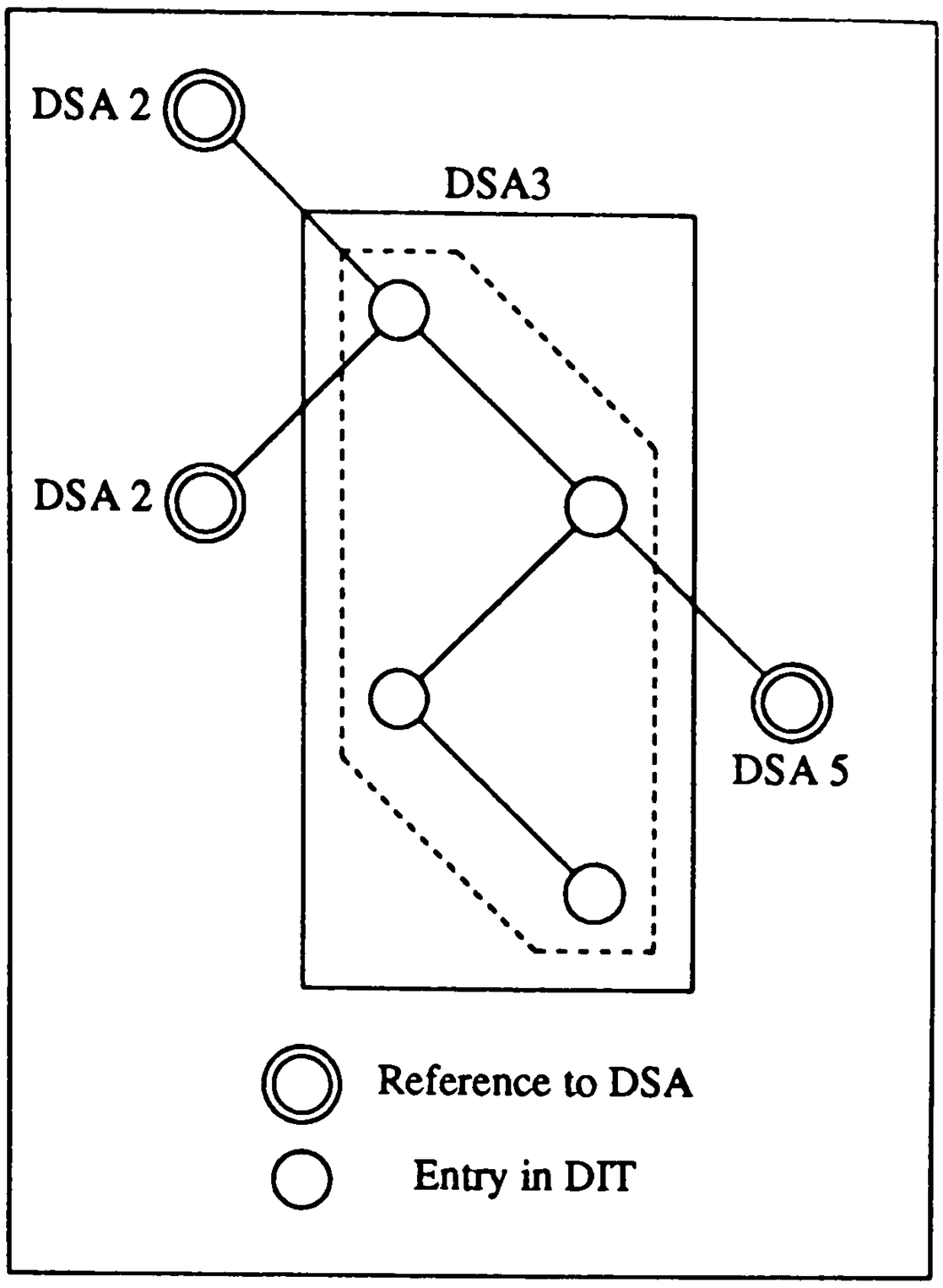


Figure 5.4: Knowledge tree for DSA 3

The minimal knowledge from this knowledge tree is represented by the following table where each row of the table specifies a reference.

Minimal knowledge for DSA 3			
Name	Address	Responsibilities	Reason
name(DSA 2)	address(DSA 2)	root, /c=b/o=d/	immediate superior
name(DSA 3)	address(DSA 3)	/c=b/	self
name(DSA 5)	address(DSA 5)	/c=b/o=e/u=i/	immediate inferior

The minimal knowledge of DSA 1 is represented by the following table.

Minimal knowledge for DSA 1			
Name	Address	Responsibilities	Reason
name(DSA 2)	address(DSA 2)	root, /c=b/o=d/	immediate superior
name(DSA 1)	address(DSA 1)	/c=a/	self
name(DSA 4)	address(DSA 4)	/c=a/o=c/u=f/, /c=a/o=c/u=g/	immediate inferior

Summary for minimal knowledge

In summary, each DSA must be able to navigate an operation closer to a responsible DSA if successful navigation is to occur. A *navigation step* can be performed if a DSA possesses *knowledge* of its immediately inferior and superior DSAs so that the operation can be navigated one step in the right direction. This knowledge is called *minimal knowledge* and consists of a set of *references* to other DSAs specifying their names, addresses and responsibilities in terms of their *fragments*.

5.2.4. Navigation step algorithm

The previous section has specified minimal knowledge, guaranteeing that a DSA can perform a navigation step. This section describes how a DSA uses its minimal knowledge to perform a navigation step as part of the navigation process.

A navigation step should determine whether the target entry of an operation is local and, if it is not, should determine the name and address of the closest known DSA to the target. The closest DSA is the one having a responsibility (root of a known fragment) best matching the target name.

The DSA determines the best match by searching its minimal knowledge for the reference having the responsibility which is the prefix of the target name matching the greatest number of name parts. For example, referring to the tables above, DSA 3 would return its own reference with responsibility */c=b/*, matching the target entry */c=b/o=c/*. The results of the search are treated in the following way:

- If a match is found, the reference indicates the name and address of the closer DSA. This may be the local DSA in which case the query is local.
- If no match is found, the DSA assumes that the target entry is in an entirely different subtree of the DIT and so navigation should proceed towards the root of the DIT, via the immediately superior DSA. In this case, the reference of the closer DSA is that of the DSA's immediate superior.
- If no match is found and this DSA is the root then the name is erroneous and a *name error* should be generated.

The navigation algorithm must also dereference aliases. If the target entry is determined as the responsibility of the local DSA then it performs *name verification* by examining its fragments. During the course of local verification the DSA might encounter an alias entry. In this case, the DSA replaces the parts of the target name matching the alias with the name pointed at by the alias to generate a new target entry name. For example, local verification of the name */c=a/o=m/u=f/p=g/* might encounter the alias */c=a/o=m/* pointing at the entry */c=a/o=c/* . This generates the new target name: */c=a/o=c/u=f/p=g/*. The new target may be non-local and so the navigation step is repeated from the beginning with the newly generated target name.

Dereferencing aliases must account for the possibility of alias loops as described within chapter 3. In particular, alias loops should be detected during navigation. This could be achieved by remembering the distinguished names of all aliases dereferenced during each navigation process. If any alias is encountered for a second time, an alias loop is detected and navigation terminates. Due to the distributed nature of navigation, this mechanism requires that the distinguished names of dereferenced aliases are transmitted between DSAs during navigation.

The navigation algorithm, used within each DSA, is summarised below:

```
For a given target name {

        Search minimal knowledge for the reference which
        has a responsibility best matching the target name

        Return a name error if no match is found and this
        is the DSA responsible for the root entry.

        Return the reference of the immediately superior DSA
        if no match is found and this is not the root DSA

        If a match was found for another DSA then return
        the matching reference

        Otherwise, the target entry is the responsibility
        of this DSA {

        Perform local name verification.

        If an alias is encountered {

                Check the list of dereferenced aliases within
                the operation to see whether it has been seen
                before. If so, terminate navigation.

                Otherwise, replace the matched name parts with
                the aliased name and restart navigation with
```

```
                    the new target name.
                }

                Otherwise return results of verification.
            }
        }
```

This concludes the specification of minimal knowledge and basic directory navigation. The following section discusses how navigation may be optimised by the use of *opportune knowledge*.

5.2.5. Opportune knowledge

A DSA's minimal knowledge specifies the minimum knowledge required for successful navigation. However, it does not necessarily specify the knowledge required for optimum navigation. Navigation may be improved by the introduction of opportune and replicated knowledge. This section discusses the structure and use of opportune knowledge.

In general, opportune knowledge serves two main purposes:

- It increases the efficiency of navigation.
- It increases the robustness of navigation.

Navigation using minimal knowledge alone involves a sequence of short hops between adjacent DSAs. For example, referring to figure 5.3, an operation on the entry */c=a/o=c/u=f/p=j/* initiated at DSA 5 would involve DSAs 5, 3, 2, 1 and 4 in navigation. Navigation would be far more efficient if DSA 5 knew of the existence and responsibilities of DSA 4 and could miss out DSAs 3, 2 and 1 in the navigation process.

The efficiency of navigation will generally be improved if DSAs hold references to other DSAs which were not their immediate superiors or inferiors. These references define opportune knowledge.

In addition to increasing the efficiency of navigation, opportune knowledge also increases the robustness of navigation. Each DSA performing a navigation step may find the references of several DSAs closer to the target entry. If the first choice DSA is unobtainable then navigation may continue via another DSA. Thus, opportune knowledge may allow an unobtainable DSA to be excluded from the navigation sequence. This use of opportune knowledge is important in reducing the effects of DSA crashes and communication failures and therefore improving the overall reliability of the Directory service.

Opportune knowledge is defined by the following simple ASN.1 structure.

```
OpportuneKnowledge ::= SET OF Reference
```

The navigation algorithm operates as specified in the previous section except that the DSA searches its opportune knowledge as well as its minimal knowledge for several matching references. In this way several DSAs might be bypassed in the navigation sequence. For example, DSA 5 might hold an opportune reference to DSA 1. This would result in the above navigation sequence being reduced to DSAs 5, 1 then 4.

Although a DSA may have a choice of possible references resulting from a navigation step, it should be noted that they all navigate the operation closer to the target entry (some closer than others). An operation is NEVER navigated further away, even in the interests of bypassing unobtainable DSAs.

Opportune knowledge optimises navigation at the expense of higher maintenance and storage costs at a DSA. In the most extreme case, each DSA might hold opportune knowledge referring to every other DSA in the Directory. However, this would result in a large update overhead whenever the configuration of the Directory system altered. There is clearly a tradeoff between better navigation performance and maintenance overheads. This chapter does not propose any limitations on the amount of opportune knowledge which may be held by a DSA . However, the following paragraphs suggest different approaches to its acquisition and use. The problems of updating knowledge following directory reconfiguration are addressed by section 5.3.

Bypassing a heavily loaded DSA

Opportune knowledge might be used to relieve the burden on a heavily loaded DSA by bypassing it in the navigation process. For example, all the immediately inferior DSAs of a busy DSA might reference each other. This is shown in figure 5.5. This use of opportune knowledge might be prohibitive if a DSA has a large number of inferior DSAs.

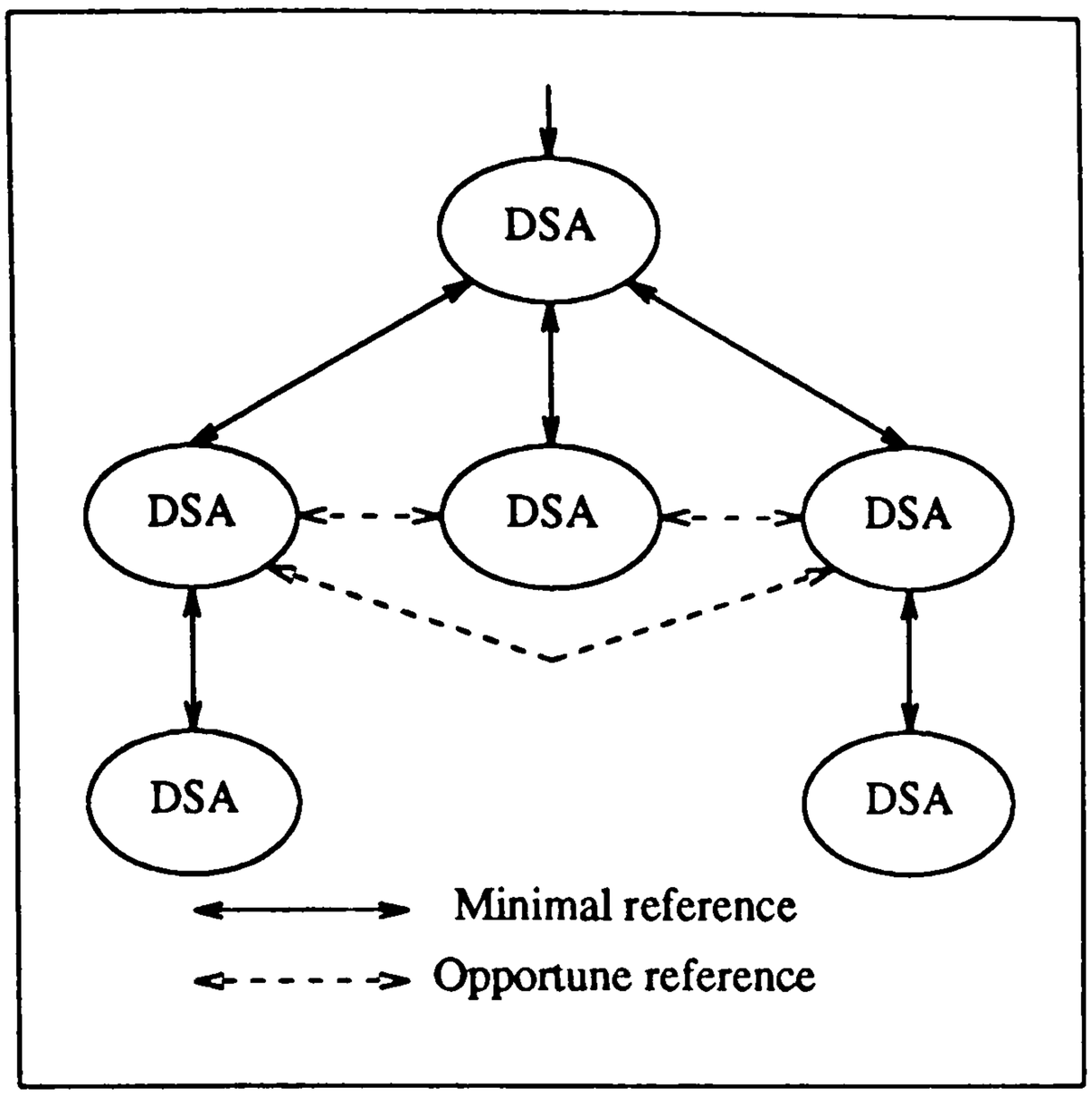


Figure 5.5: Opportune knowledge relieving a loaded DSA

Direct navigation to major DSAs

Opportune knowledge might also be used to direct operations to a few strategically placed DSAs allowing more direct routing to distant parts of the Directory information tree. For example, it might be useful if a DSA were to reference all *country-level* DSAs. However, the widespread use of opportune knowledge in this manner could place heavy burdens on certain DSAs. This use of opportune knowledge is depicted in figure 5.6.

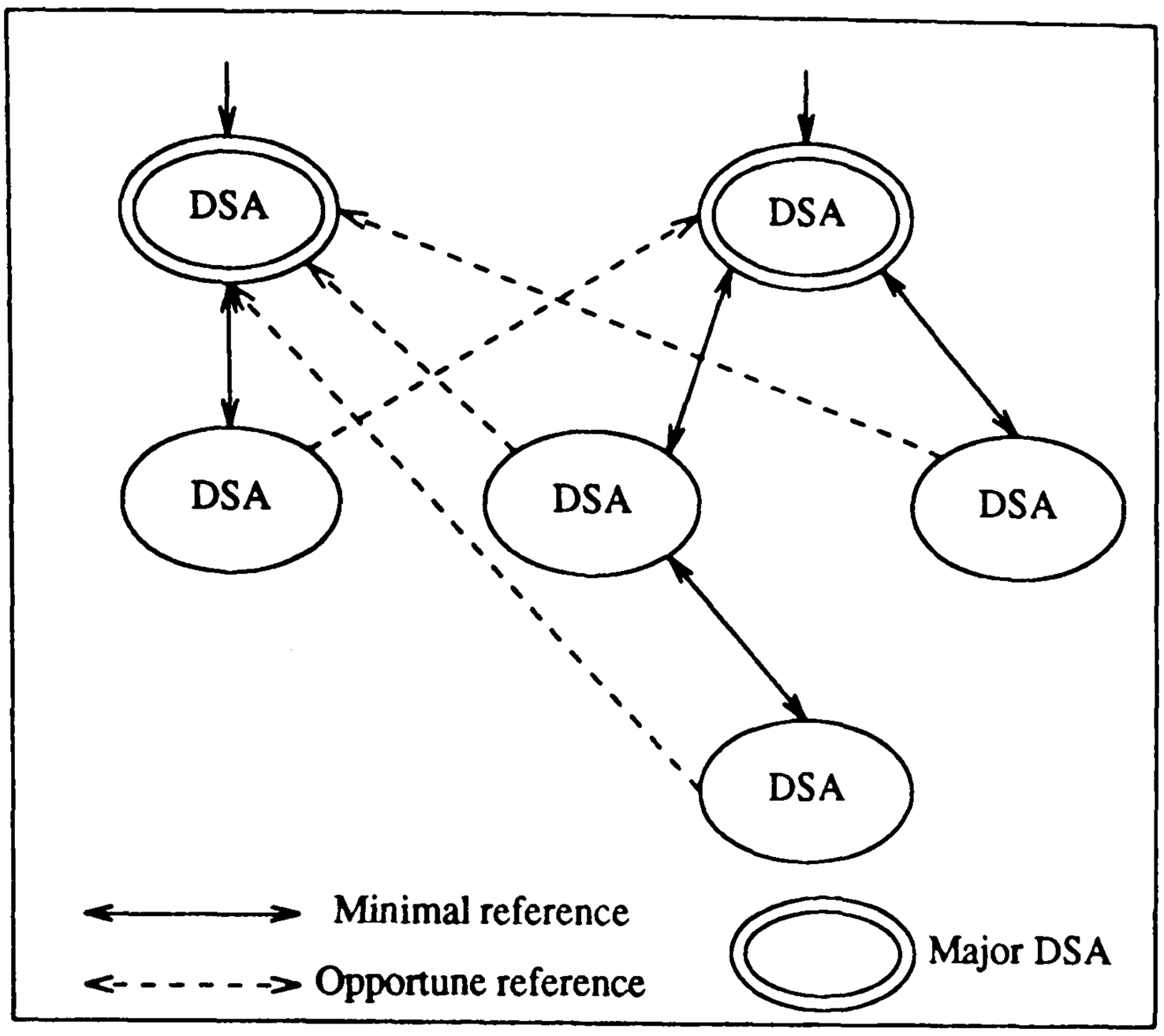


Figure 5.6: Opportune knowledge - routing via major DSAs

Projects involving different organisations

Projects spanning different organisations may involve frequent communication between their members. Navigation may be aided if each organisation's DSA references the other for the duration of the project (however optimisation could also be achieved by *replication* methods).

Caching

The example uses of opportune knowledge given above describe static situations where administrators might optimise navigation to overcome a recognised problem or bottleneck. It would be useful if the Directory also supported more dynamic use of opportune knowledge. Ideally, DSAs might acquire and maintain their own opportune references without the intervention of human administrators. This could be supported by caching techniques.

A DSA might choose to cache references to other DSAs obtained during routing. For example, DSAs could pass their references along a chain during *chaining*, allowing caching by other DSAs. Alternatively, a DSA coordinating *referral* could swap references with each

DSA it contacted and could cache the results.

Successful caching is likely to involve *time to live* (TTL) timestamps on cached references. These specify the valid lifetime of information and enable out of date references to be deleted. The selection of TTL timestamps will be a critical choice and may vary considerably. In general, TTL will depend on the rate of changes to DSAs responsibilities (expected to be relatively low) and the rate of opportunities to refresh references (possibly quite high). It may be quite feasible to set TTLs to days or even weeks once a working directory is established although references should be refreshed at every available opportunity.

In addition to the direct caching of information, DSAs could support *negative caching*. Negative caching occurs when a DSA records that certain information is not present at another DSA. This might happen following a query which failed to retrieve information and could be used to prevent the query being tried again, thus improving efficiency.

The caching of references to form opportune knowledge provides an easy way for the Directory system to be partially self-configuring without the need for a complicated administrative protocol. Knowledge of new DSAs or changes in DSA's responsibilities will percolate throughout the Directory system as operations are sent from DSA to DSA. This assumes that DSAs include their references in operation requests and results (see section 5.4.1).

Each DSA can determine its own caching policy and quite complex techniques can be envisaged which examine the rates of access to other DSAs, monitor for bottlenecks and adjust minimal knowledge accordingly. The specification of such methods is not a part of this thesis. However, the gathering and analysis of performance statistics is part of general systems management and, consequently, this work could provide useful input to the implementation of caching mechanisms.

It should be noted that caching techniques are particularly susceptible to inconsistencies introduced as a result of directory reconfiguration. A DSA using cached references must therefore be prepared for, and react gracefully to, possible inconsistencies in opportune knowledge. This is discussed in section 5.3.

5.2.6. Summary of partitioning, knowledge and navigation

This section has examined how the Directory Information Tree may be divided between the DSAs constituting the Directory system. It has also described how navigation can locate named entries without the need for centralised control.

Each DSA is responsible for one or more *fragments* of the DIT consisting of incomplete subtrees identified by the name of their root entry. The list of fragment names belonging to a DSA determine the DSA's responsibilities and are described by a structure called its

reference.

The process of navigating operations through the Directory system requires that each DSA can perform a *navigation step* to determine whether a target entry is local and, if not, the name of a closer DSA to the target. A DSA performs a navigation step by inspecting its *knowledge* of other DSAs' responsibilities.

Knowledge is structured as a set of references and may be subdivided into *minimal* knowledge, essential for navigation to occur, and *opportune* and *replicated* knowledge, used to optimise and increase the robustness of navigation. This section has considered how opportune knowledge may be acquired and under what circumstances it may be used.

The next section discusses the problem of managing knowledge as the Directory system is reconfigured. It shows that the navigation procedure must cope with inconsistent knowledge and describes how this may be achieved. It also considers mechanisms for automatically correcting knowledge inconsistencies.

5.3. Knowledge management

The previous section introduced the concept of knowledge, supporting the navigation of operations, and noted that the three categories of knowledge described above are subject to different management mechanisms.

Knowledge is the glue binding DSAs together and is vital to the operation of the Directory. Knowledge management is therefore of major importance. This section considers the management of knowledge as the configuration of the Directory system changes and addresses many of the management issues described in section 5.1.

- Section 5.3.1 outlines the problems of managing knowledge information as the configuration of the Directory system changes. It explains how knowledge inconsistencies arise and the effect they may have on navigation.
- Section 5.3.2 describes the management of minimal knowledge and how it must be updated to guarantee connectivity between DSAs.
- Section 5.3.3 discusses the management of opportune knowledge. It specifies mechanisms allowing navigation using inconsistent opportune knowledge. It also considers the propagation and automatic correction of knowledge.

A major goal of this section is to specify the automatic management of knowledge with minimum human intervention.

5.3.1. The effects of directory reconfiguration

The *configuration* of the Directory system describes the partitioning of the DIT between DSAs. This configuration may change over time as DSAs are introduced or their responsibilities are restructured. Reconfiguration of the Directory might involve the following changes.

- New DSAs might be added to the system
- DSAs might be removed from the system
- DSAs might alter their responsibilities.

These changes may have several effects on the knowledge in the Directory system. The introduction of a DSA requires that other DSAs add its reference to their knowledge. Conversely, the removal of a DSA will require the deletion of references from knowledge. Altering a DSA's responsibilities or address will require that references are updated.

Knowledge management describes the problem of maintaining knowledge in an environment where DSAs are changing. Without adequate knowledge management, inconsistences may arise resulting in the incorrect operation of the distributed system. These knowledge inconsistencies involve inaccurate or missing references and may have consequences ranging from loss of performance in navigation to loss of connectivity between DSAs. They may even result in infinite navigation loops. The following paragraphs briefly describe possible knowledge inconsistencies and their consequences.

- A missing reference may result in several DSAs being disconnected from the remainder of the system due to a connecting DSA being unreachable.
- A reference to a non-existent DSA will result in a binding error when an attempt is made to connect to it.
- A reference describing an existing DSA with incorrect responsibilities may have several effects. In a simple case, it may result in a loss of performance as an operation is navigated further away from a responsible DSA. In a pathological case, it might result in a navigation loop. This would involve navigating an operation to a DSA which had already seen it, thus causing an infinite loop.

Figure 5.7 shows a simple navigation loop caused by an incorrect reference in opportune knowledge.

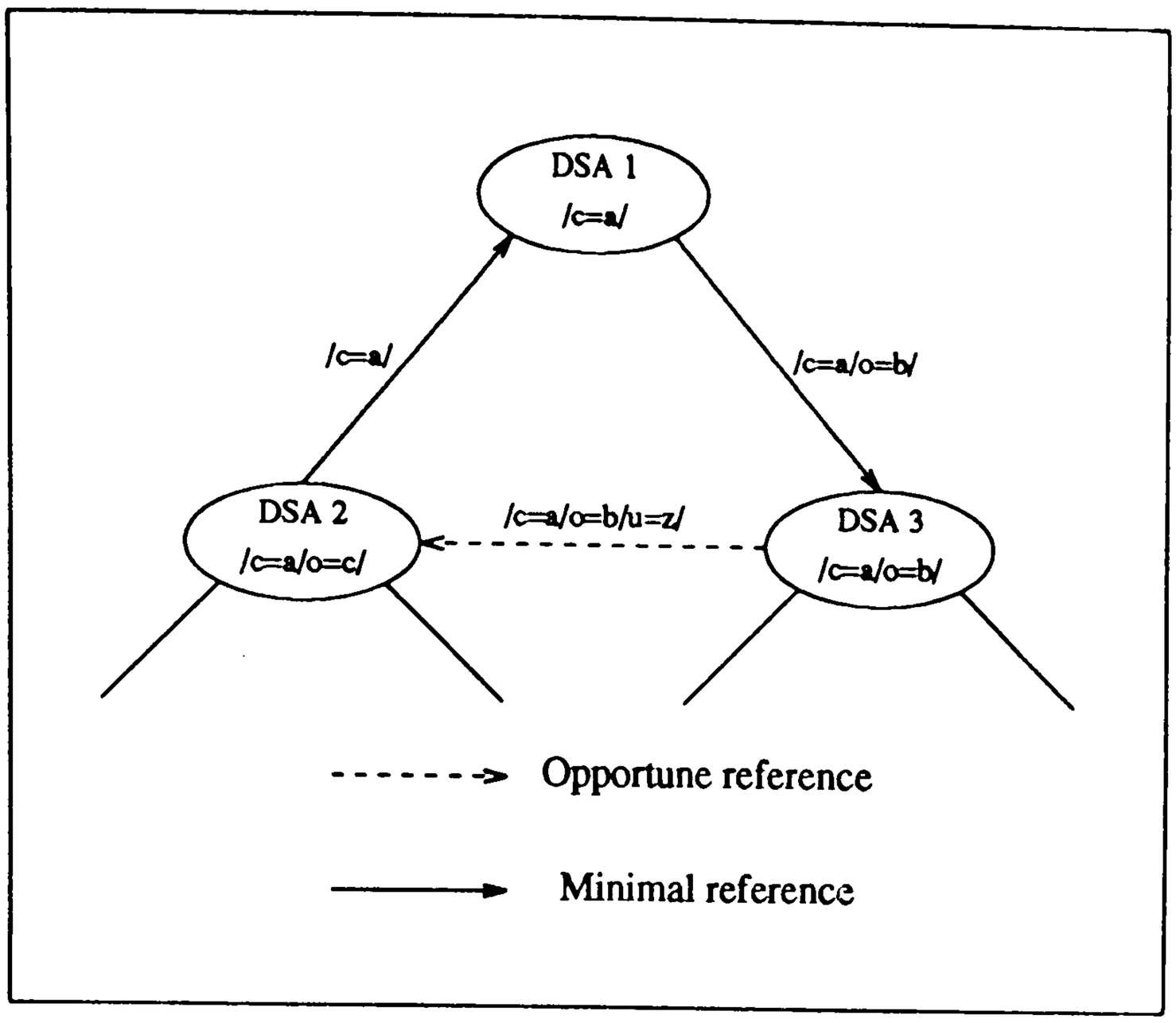


Figure 5.7: A simple navigation loop

This may have been caused by a change in responsibility for DSA 2 without an update of the opportune reference at DSA 3. If an operation accessing the entry */c=a/o=b/u=z/* were directed at any of the DSAs in the figure the result would be an infinite navigation loop.

The following sections describe how knowledge may be updated following system reconfiguration and how pathological navigation errors may be avoided. They consider the separate issues of the management of minimal and opportune knowledge and show that they have different requirements and solutions.

5.3.2. Management and consistency of minimal knowledge

Minimal knowledge guarantees that navigation can occur and specifies the fundamental connectivity between DSAs. If minimal knowledge suffers from inconsistencies then the correct operation of the Directory can no longer be guaranteed. Steps must therefore be taken to ensure that minimal knowledge is maintained in a consistent state as reconfiguration occurs. This requires that changes to DSA responsibilities and addresses are ALWAYS propagated to the relevant minimal knowledge of other DSAs. Fortunately, this will typically involve a small number of DSAs for any one change.

Minimal knowledge references to a DSA are only maintained by well known, immediately adjacent, DSAs and, therefore, update of a DSA's responsibilities will only affect minimal knowledge in a limited area of the Directory. For this reason, it is possible to consistently manage minimal knowledge without an expensive and complex update procedure. The following examples indicate the knowledge management steps which need to be taken when adding, removing and altering DSAs.

Adding a leaf DSA

The addition of a new leaf DSA (one having no inferior DSAs) requires coordination between the new DSA and its immediately superior DSA.

- The superior DSA adds the new DSA's reference to its minimal knowledge.
- A DIT entry is created naming the new DSA.

The new DSA can then "boot up" so long as its minimal knowledge includes its own reference and the reference of its superior DSA. When booted, the new DSA should check that minimal knowledge is consistent between itself and its superior. This thesis proposes that the checking and updating of references between DSAs can be achieved by the introduction of a new *Update Reference* operation used by one DSA to update its reference at a target DSA. This operation also retrieves the reference of the target DSA. The Update Reference operation belongs to the Directory System Protocol and is described by the following ASN.1 definition.

```
UpdateReference ::= ABSTRACT-OPERATION
                    ARGUMENT UpdateReferenceArgument
                    RESULT UpdateReferenceResult
                    ERRORS { ReferenceError, DSAError }

UpdateReferenceArgument ::= SET {
                            old [0] Reference,
                            new [1] Reference }

UpdateReferenceResult ::= Reference
```

This operation is called by one DSA at another DSA. Providing that the calling DSA is the same as identified within the supplied references, the receiving DSA must replace its old reference with the new. The receiving DSA then returns its own reference to the calling DSA.

Removing a leaf DSA

The removal of a leaf DSA requires coordination between the DSA to be removed and its immediately superior DSA. The DSA to be removed should take the following steps.

- Delete its reference from its immediately superior DSA.
- Delete its name from the DIT.

The DSA can then be shutdown. This thesis proposes the *Delete Reference* operation to remove the reference of a DSA at another DSA. It is specified by the following ASN.1 structure.

```
DeleteReference ::= ABSTRACT-OPERATION
                    ARGUMENT DeleteReferenceArgument
                    RESULT DeleteReferenceResult
                    ERRORS { NavigationError, DSAError }
                                                    •

DeleteReferenceArgument ::= Reference

DeleteReferenceResult ::= NULL
```

It should be noted that the operations described above are not controlled by the directory access control mechanism. This is because they do not belong to the global conceptual model. However, they may only legally be called by a DSA to maintain its own reference at another DSA. Furthermore, the target DSA must perform them. This requires the mutual authentication of DSAs by the Authentication service. The issue of access control between DSAs is briefly revisited in section 8.6 of this thesis.

Changing a DSA's responsibilities

Consider the case where a DSA (DSA1) assumes responsibility for a fragment of the DIT previously held by another DSA (DSA2). The following steps need to be taken by DSA1 and DSA2 to ensure that minimal knowledge remains consistent.

- Firstly, they must update their own references to reflect their new responsibilities (DSA1 adds the new fragment and DSA2 removes it).
- Secondly, they must update their references at all adjacent DSAs. These DSAs are those listed in the minimal knowledge of DSA1 and DSA2 and each one can be updated by an Update Reference operation as described above.

Thus, DSAs 1 and 2 need to apply the Update Reference operations to adjacent DSAs. Overall synchronisation between DSA1 and DSA2 is the responsibility of their administrators. However, only DSA1 and DSA2 need to be synchronised. All other affected DSAs will receive updates automatically from these two.

The case where a DSA assumes responsibility for a newly created fragment or changes its address is a trivial instance of the "two DSA" case. It can therefore be handled by the simple use of the Update Reference operation to update all adjacent DSAs.

Summary of minimal knowledge management

Minimal knowledge must be maintained in a consistent state if navigation is to function correctly. The above examples show how minimal knowledge can be managed with only a minimum of cooperation between DSAs and still remain in a consistent state. In each case, updates can be propagated and checked by the affected DSAs using the *Update Reference* and *Delete Reference* operations.

It is important to note that a DSA can always determine which other DSAs require updating from its minimal knowledge alone.

5.3.3. Management and consistency of opportune knowledge

The previous section discussed the management of minimal knowledge and noted the requirement that it be maintained in a consistent state. This section discusses the management of opportune knowledge. It shows that maintaining the consistency of opportune knowledge is extremely difficult and therefore describes mechanisms for coping with and curing inconsistencies when they are encountered.

Section 5.2.5 indicated that opportune knowledge may be acquired by ad-hoc methods such as caching. The common feature of these methods is that the DSA storing the opportune reference assumes the responsibility for its acquisition. The referenced DSA may not even be aware that its reference has been stored. The responsibility for managing opportune references therefore passes from the referenced DSA (as is the case with minimal knowledge) to the DSA holding the reference.

In general, a DSA cannot determine which other DSAs hold opportune references to it. It is therefore not practicable for DSAs to be certain that opportune references are accurate at all times. Furthermore, even if it were, the potentially large volume and diversity of opportune knowledge would result in a high maintenance overhead.

It follows that the Directory system cannot support a strict protocol for the consistent update of opportune knowledge as it does with minimal knowledge. This implies that the Directory must be able to function in the presence of inaccurate opportune knowledge, providing that accurate minimal knowledge is available. In particular, the navigation procedure must cope gracefully with inconsistent opportune knowledge.

The remainder of this section specifies mechanisms supporting navigation using possibly inconsistent opportune knowledge. A major goal of these mechanisms concerns the detection and automatic correction of knowledge inconsistencies.

Navigating with inconsistent knowledge

The navigation procedure can be extended to deal with possible inconsistencies in opportune knowledge by implementing the following steps:

1. Navigation begins using both opportune and minimal knowledge.*
2. If a loop or incorrect opportune reference is detected then navigation continues using only minimal knowledge. The inconsistency may also be automatically corrected.
3. If a loop or incorrect reference is detected when using minimal knowledge then navigation is terminated. Furthermore, the error could be logged or reported to a DSA administrator for future correction.

Minimal knowledge should always be consistent due to the management operations specified by the previous section and, therefore, steps 1 and 2 should be sufficient to ensure navigation. However, nothing is certain, and so step 3 provides an additional safeguard to protect against corrupted minimal references.

The extended navigation algorithm described above requires that DSAs are able to detect loops and erroneous references. Furthermore, this should be possible using only local information in order to preserve DSA autonomy. This thesis proposes that error detection can be achieved by including the following three mechanisms in the directory navigation algorithm.

- A *hop-count* on operations.
- Including the assumed *intended reference* in each operation directed at a DSA.
- Each DSA involved in the navigation sequence checking that it is closer to the target entry than the last (*route checking*).

These mechanisms are described below and all three should be adopted by the directory navigation procedure. Furthermore, in order for these algorithms to be effective, they must be adopted as standard by all DSAs in the Directory system.

---

*It should be possible for users to limit navigation to use only minimal knowledge if they wish.

Hop-count

It is possible to determine the maximum number of DSAs which should be encountered during each navigation process by inspecting the target entry name and responsibilities of the initial DSA receiving the operation. The longest navigation route that should be taken is from the initial DSA to the root of the DIT and then from the root to the target entry. This would use only minimal knowledge.

In the worst case, the operation would be navigated through a sequence of hops each consuming one vertex of the DIT. The maximum number of DSAs which can be encountered on this route is equal to the sum of the number of name parts in the responsibility of the initial DSA and the name of the target entry. For example, referring to figure 5.3, the maximum number of DSAs that could be encountered in navigating to the entry */c=a /o=c /u=f/* from DSA 5, responsible for the fragment */c=b /o=e /u=i/*, is 6 no matter how the rest of the DIT is partitioned.

Simple loop control may therefore be achieved by setting a *hop-count* field in the operation heading, decremented at each new DSA encountered in the navigation process. This field is initialised by the initial DSA in the navigation sequence and represents the maximum number of DSAs which should be encountered on the route to the target DSA. If the hop count becomes negative a navigation error has occurred.

Dereferencing aliases may cause the structure of a name to alter drastically. In particular, the number of attributes in the name could increase. For this reason, whenever an alias is dereferenced, the hop-count should be reset to reflect the new length of the purported name.

It should be noted that the use of opportune references to bypass unobtainable DSAs in the navigation process does not require the hop-count to be reset because alternative references are only used if they navigate the operation closer to the target entry (section 5.2.5).

It should also be noted that the hop-count method will not prevent an alias loop. However, alias loops should be detected during the general navigation step procedure (see section 5.2.4).

The hop-count method of loop control is guaranteed to prevent an operation from following an infinite navigation loop but fails to indicate where the inconsistency is located or to correct it. The hop-count is therefore a simple but crude last resort for navigation loop control.

Intended reference

A more informative method of loop control may be implemented where each DSA contacting a second DSA during navigation includes the assumed reference of the second in an *intended DSA* argument to the operation. The second DSA can then check whether this intended reference is the same as its true reference. If it is not, an error has been detected in the knowledge of the first DSA. Furthermore, the second DSA might then update its reference at the first using the *Update Reference* operation, thus correcting the error.

This method of loop control has several advantages over the hop-count method: The navigation error is detected when it occurs, the location of the error is known and the error may be corrected automatically.

Route checking

Each DSA receiving an operation from a first DSA may check whether it is in fact closer to the target entry as a result of the last navigation step. This is possible if a *last DSA* argument to the operation includes the reference of the last DSA to perform a navigation step. Where the DSA referral mode of operation is supported, the *last DSA* need not be the DSA calling the operation.

It is possible to determine which of the two references (last DSA and current DSA) is closer to a target name by counting the number of name parts of their responsibilities matching the target name. For example, if m is number of name parts in the best match responsibility of reference A and n is number of name parts in the best match responsibility of reference B then A is closer to the target name than B if:

m > n (case 1) or

m = n = 0 and the responsibility of A is superior in the DIT to that of B (case 2)

In the second case the operation is navigating towards the root of the DIT and A is closer to the root than B. For example, referring to figure 5.3, DSA 3 with reference */c=b/* is closer to the entry */c=b /o=e /u=i/* than DSA 2 which references the root (by case 1). This is in turn closer than DSA 1 with reference */c=a/* (by case 2)

This method of loop control also determines the location of inconsistent knowledge and may allow consistency to be restored as in the previous method.

In summary, the navigation procedure is able to adapt to inconsistencies in opportune knowledge by detecting reference errors and loops and switching to the exclusive use of minimal knowledge. Errors may be detected by a combination of the hop-count, intended reference and route checking mechanisms described above. All three methods are used

simultaneously to reduce the chances of failure.

As noted above, these mechanisms must be adopted by all DSAs if they are to be effective. This requires standard support from the Directory Service Protocol, particularly for including references within operation arguments. This support is specified in section 5.4.1.

The Directory system may also automatically correct inconsistencies in opportune knowledge using some of the above mechanisms and the *Update Reference* operation.

The conclusion of this work is that opportune knowledge cannot support an explicit management protocol. Instead, the Directory will allow, and then correct, inconsistencies as they are encountered. It should be noted that *time to live* information on cached references will also facilitate the management of opportune knowledge by allowing out of date references to be removed.

5.3.4. Summary of knowledge management

This section has described the management of knowledge, meaning the update of knowledge to reflect system reconfiguration. Knowledge management is vital to ensure the correct operation of the navigation procedure and updates should be managed by the Directory system with minimal human intervention. The previous paragraphs made a distinction between the management of minimal and opportune knowledge and have proposed different solutions to accomplish these tasks.

Minimal knowledge is essential to ensure the fundamental connectivity of the Directory system and must always exist in a consistent state. The previous sections have described the steps necessary to maintain minimal knowledge and have specified operations for this purpose. The use of a strict protocol is possible because only a small number of known DSAs are likely to require updating at one time.

The use of a similar strict protocol to manage opportune knowledge will not be feasible due to its diversity and volume. Instead, opportune knowledge will be managed by ad-hoc methods and the navigation procedure must therefore be able to cope with incorrect opportune references, leading to errors and even loops. This is possible via a combination of error detection mechanisms and these may often be used to automatically correct inconsistent references. This work has emphasised the role of the Directory system in managing knowledge without the need for human intervention.

The next section considers the distributed execution of directory operations. It describes the DSA interactions necessary to implement user requested operations. It also discusses the Directory System Protocol including distributed support for the directory access and integrity control mechanisms.

5.4. Distributed operation of the Directory

This section examines the distributed execution of the abstract operations specified by chapter 3 within a system of cooperating DSAs. Each abstract operation may require the interaction of several DSAs and a major goal of this section is to specify how these interactions occur. This includes describing the distributed support required for the directory access control and data integrity mechanisms.

Specifying the distributed operation of the Directory fulfils the same goal as examining *query processing* within more general database systems. However, the Directory service does not support a general, high-level query language such as *SQL* and so its query processing is far simpler.

The table below lists the abstract operations defining the Directory Access Protocol as proposed by chapter 3 of this thesis. It also notes whether they access single or multiple target entries.

DAP abstract operations	
Operation	Target entries
Read Entry	single
List Subordinates	multiple
Search	multiple
Modify Entry	single
Add Entry	single
Delete Entry	single
Add Alias	single
Delete Alias	single
Suspend Entry	single
Reinstate Entry	single
Attribute/Entry Definition ops	single

DSA interactions will be specified in terms of new abstract operations forming the *Directory System Protocol* (DSP).

This remainder of this section is structured in the following way.

- Section 5.4.1 describes the general structure and execution of DSP operations including support for navigation control and knowledge management as described above.
- section 5.4.2 discusses those operations accessing single entries and specifies mechanisms for realising efficient access control and integrity checking.

- Section 5.4.3 discusses the *list* and *search* operations accessing multiple entries.
- Section 5.4.4 discusses the distributed management of entry and attribute definitions.
- Section 5.4.5 compares the chaining and referral models of directory operation.

A major goal of this section is reducing communication costs and delays during the execution of operations. This is achieved by minimising the number of DSA interactions throughout the course of each operation under the assumption that DSA interactions will represent significant delay and cost.

The following work assumes that all DSAs referencing each other are, in fact, interconnected. In reality, this may not be the case for several reasons:

- Firstly, DSAs may be unobtainable due to failures in DSA software, hardware or communication links. This problem concerns *crash recovery* and is discussed within chapter 6.
- Secondly, the assumption that all DSAs will cooperate may be incorrect. In many cases, DSAs may not trust each other sufficiently to support interaction. This problem concerns system level access control and is discussed in section 8.6.

For the time being, this thesis assumes possible interconnection between all DSAs.

5.4.1. General structure and execution of distributed operations

This section describes the general structure and execution of distributed operations. It specifies *operation envelopes* and *result envelopes* representing navigation information within the common arguments and results of operations.

Although some abstract operations may affect multiple entries in the DIT, every operation can be directed at a single initial target entry. For example, List Subordinates names a parent entry and Search names the root of a subtree. The distributed implementation of an operation therefore involves the following two phases.

- A *navigation* phase involving routing to the DSA responsible for the initial target entry. This DSA is called the *coordinating DSA*.
- An *execution* phase where the operation accesses a number of entries at several DSAs. This may require DSA interactions controlled by the coordinating DSA.

The navigation and execution phases of a general operation are shown in figure 5.8 and are described by the following sections.

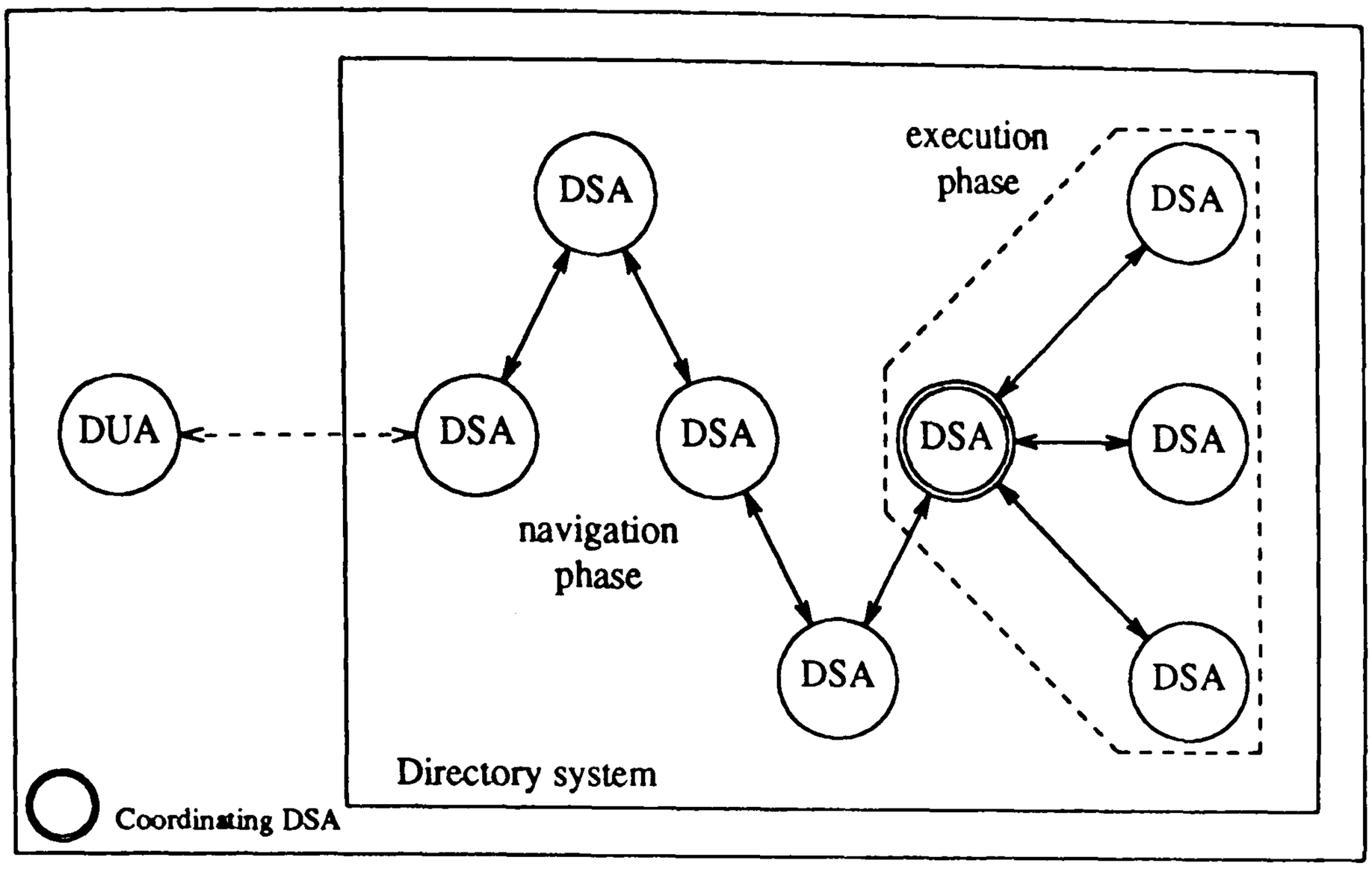


Figure 5.8: Navigation and execution phases of an operation

The navigation phase and operation envelopes

The navigation phase is similar for all operations. The user requested operation is mapped into a DSP operation including additional navigation information. The new operation is then navigated to the coordinating DSA. Navigation might use the chaining or referral modes of directory operation.

The mapping from the DAP to the DSP versions of an operation can be viewed as constructing an *operation envelope* which is an additional argument to the operation (additional to the user specified arguments) including the information used for navigation control and error detection. The structure of an operation envelope is the same for all DSP operations. Furthermore, the results of operations are given a common *result envelope* having a similar purpose. These structures also support the propagation of opportune knowledge.

The following ASN.1 structure defines an operation envelope.

```
OperationEnvelope ::= SET {
        minimal-only [0] BOOLEAN DEFAULT FALSE,
        hop-count [1] INTEGER,
        intended-dsa [2] Reference,
        last-dsa [3] Reference,
        sending-dsa [4] Reference,
        aliases-seen [5] SET OF DistinguishedName }
```

The elements of this structure have the following meanings: The *minimal-only* flag may be set to indicate that navigation should proceed using minimal knowledge only. The *hop-count* field contains the navigation hop count, decremented at successive DSAs. The *intended-DSA*, *last-DSA* and *sending-DSA* elements contain the references of DSAs associated with a particular navigation step. The intended-DSA is the one receiving the operation, the sending-DSA is the one sending the operation and the last-DSA is the last DSA to perform a navigation step. The distinction between the sending-DSA and last-DSA is required during the DSA referral mode of operation. Finally, the *aliases-seen* element contains the names of aliases already dereferenced during the navigation process. This information is required for detecting alias loops.

The following ASN.1 structure defines a result envelope.

```
ResultEnvelope ::= SET {
                    result-type [0] BITSTRING,
                    sending-dsa [1] Reference,
                    intended-dsa [2] Reference }
```

The elements of this structure have the following meanings: The *result-type* element indicates whether any non-fatal navigation errors occurred and may indicate whether a DSA needs to correct its opportune knowledge. The other elements contain the references of the DSAs involved in a navigation step and are used for detecting loops and knowledge inconsistencies as described in section 5.3.3 as well as for swapping references to increase opportune knowledge.

The execution phase

The execution phase of the operation describes its implementation by a set of responsible DSAs. The coordinating DSA, located by the navigation phase, assumes control of the operation and is responsible for coordinating communication with other DSAs to obtain the results.

Many operations such as *Read Entry* and *Modify Entry* are clearly *single entry* operations where the coordinating DSA is often the only DSA involved in the execution phase. However, operations such as *Search* may involve several DSAs during the execution phase. The execution phase must also implement distributed access and integrity controls. The execution phases of different types of operation are discussed in the following sections. A major goal of these sections is to specify mechanisms ensuring that all DSA interactions during an execution phase are *single hop* and do not require further navigation, thus reducing delays and costs. This requires that all DSAs interacting in the execution phase of an operation know each other's references.

5.4.2. Single entry operations

The previous section described distributed operations in terms of a *navigation phase* and an *execution phase*.

Most directory operations access a single entry (see the table above). These single entry operations have a very simple execution phase where the coordinating DSA performs a local operation before returning the results. For example, the coordinating DSA for a *Read Entry* operation has a local copy of the entry to be read. However, the introduction of access and integrity controls complicates this situation. The support required for distributed access and integrity control is described by the following paragraphs.

Distributed support for access control

Operations are subject to access control as specified in chapter 4. Each single entry operation may require checking one *entry level* and several *attribute level* Access Control Lists for permission to perform the operation. Checking an Access Control List might require the following steps.

1. Checking that the user requesting the operation belongs to a specified subtree of the DIT.
2. Checking that the user's entry matches a specified filter.

The first of these steps may be achieved by examining the name of the subtree to see whether it is a prefix of the name of the user. This may be performed at the coordinating DSA, providing that the user's name is included in the operation envelope. It follows that, given this extension of operation envelopes, checking subtree membership does not involve any DSA interactions.

The second step requires comparison between the filter and the user's entry. There are a number of possible approaches to this:

- The contents of the user's entry might also be included in the operation envelope allowing checking to occur at the coordinating DSA.
- The coordinating DSA might read attributes from the user's entry and perform local checking. This requires a distributed read operation at the user's *home DSA* (the one responsible for their entry).
- The coordinating DSA might send the filter to the user's home DSA and request that checking occurs there. This also requires DSA interaction.

Choosing between these approaches depends on a number of factors such as communication costs and delays. However, the following points should be noted.

A user's entry may easily contain a large amount of information, particularly where graphical data is present (e.g. a photograph). It could be extremely expensive to include this in an operation envelope navigated between many DSAs.

Both reading the user's entry and transmitting the filter to the user's home DSA involve distributed communication. In either case, locating this DSA may involve time consuming navigation. In general, it is likely that transmitting the filter will be cheaper than reading the entry due to the size of the structures involved.

This thesis proposes that the first method will be too expensive in terms of communication costs and that either of the two latter methods could be adopted depending on the policies of particular DSAs. Filter checking at the user's home DSA can be supported by the introduction of the following new DSP operation.*

```
CheckFilter ::= ABSTRACT-OPERATION
                ARGUMENT CheckFilterArgument
                RESULT CheckFilterResult
                ERRORS { NavigationError, DSAError }

CheckFilterArgument ::= SET {
                        user [0] DistinguishedName,
                        filter [1] Filter }

CheckFilterResult ::= BOOLEAN
```

This operation passes a user's name and a filter to a remote DSA (their home DSA) which returns an indication of whether the filter is satisfied or not.

The major problem with checking filters is that the coordinating DSA has to navigate to the user's home DSA. This overhead can be drastically reduced by including the reference of the user's home DSA in the operation envelope along with the user's name. Navigation from the coordinating DSA to the user's DSA then only requires one step.

The operation envelope structure has now been extended to include the following additional user information:

```
UserInformation ::= CHOICE {
public [0] NULL,
identified [1] SEQUENCE {
        user DistinguishedName,
        users-dsa Reference } }
```

---

*For reasons of clarity, operation envelopes and results will not be shown as new DSP operations are specified.

This information may indicate that the user is *public* and should be granted public access to information.

Referencing the user's DSA in an operation envelope requires that the initial DSA for an operation knows its reference. This may be acquired whenever a user binds to the Directory.

Distributed support for integrity control

Operations such as *Modify Entry* and *Add Entry* may require that the coordinating DSA checks a number of entry and attribute definitions (see section 4.3). It is likely that the rate of checking definitions will be far greater than that of adding and deleting them. This section therefore proposes that entry and attribute definitions are replicated at all DSAs needing to check them. These are all the DSAs responsible for fragments within the scope of the definitions. For example, the entry definition with scope */C=GB /O=Nott.Uni/* would be replicated at all DSAs responsible for any entries in the subtree */C=GB /O=Nott.Uni/*.

Entry and attribute definitions are therefore replicated downwards within the subtree representing their scope. This replication of definitions means that the coordinating DSA will always have local copies of any entry and attribute definitions requiring checking during a single entry operation. Thus, checking integrity constraints will not require any DSA interactions. On the other hand, there is an increase in the complexity of managing these definitions. This is discussed in section 5.4.4.

Overall execution of single entry operations

In summary, a single DSA operation may be navigated to the coordinating DSA which executes it. Executing may be local to the coordinating DSA unless a filter has to be checked by the access control mechanism. This will require interaction with the user's home DSA. Navigation to this will only require a single hop providing that its reference is included in the operation envelope.

The navigation and execution phases of single entry operations are shown in figure 5.9 below.

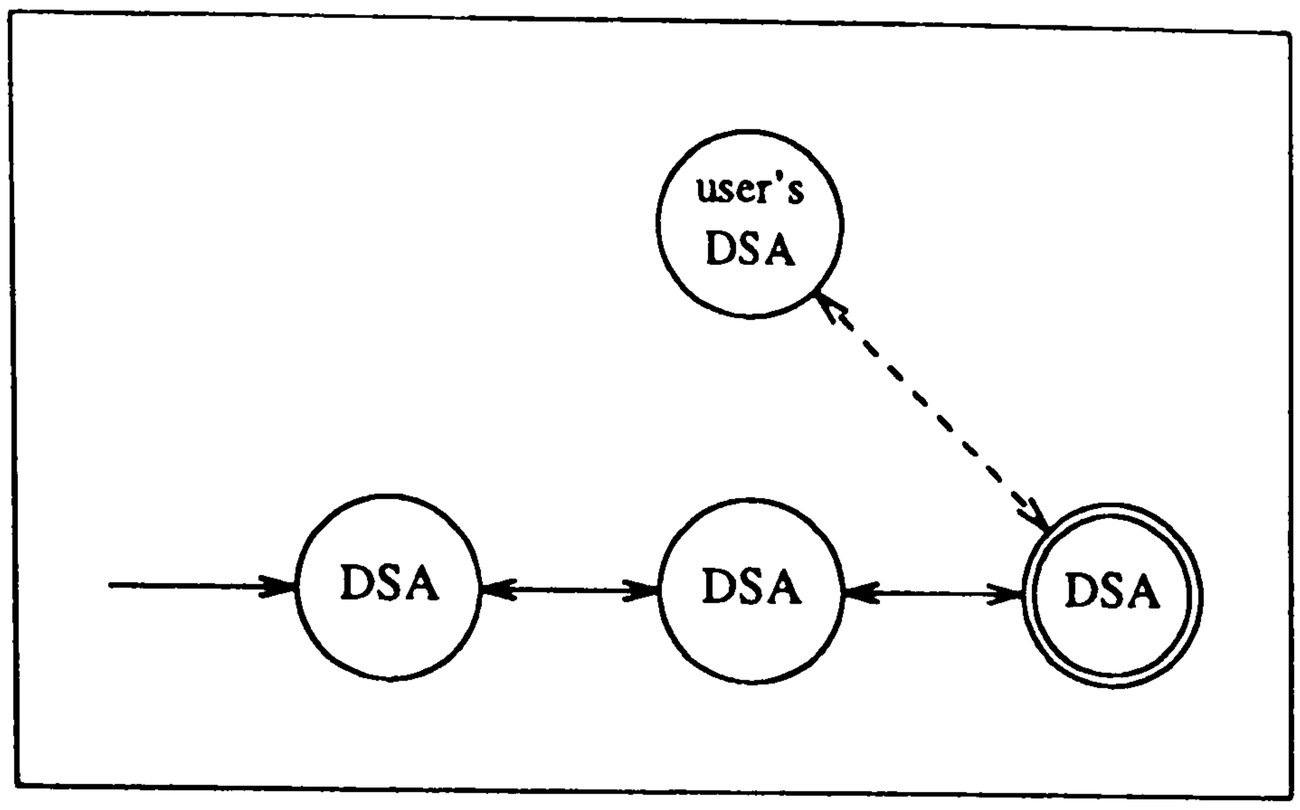


Figure 5.9: Navigation and execution phases of single entry operations

In diagrams of this sort, solid arrows indicate general DSA interactions, dashed arrows indicate possible DSA interactions and the coordinating DSA is identified by a "double circle".

Binding to the Directory

The distributed access control mechanism requires that the reference to a user's home DSA is included in the envelope of each operation they request so that filters may be checked efficiently. This section describes the steps necessary to ensure that this information is always present.

A user's *home DSA* is the DSA responsible for their entry. Users will not always access the Directory via their home DSA and therefore the initial DSA for an operation may not automatically know the reference of the user's home DSA. This reference can be obtained when the user first binds to the Directory.

Whenever a user binds to the Directory, they will be asked to supply their name. Following authentication, the initial DSA will request a *Retrieve Reference* operation which navigates to the user's home DSA using their name. The home DSA then returns its reference to the calling DSA.

The following ASN.1 structure defines the proposed Retrieve Reference operation.

```
RetrieveReference ::= ABSTRACT-OPERATION
                ARGUMENT RetrieveReferenceArgument
                RESULT RetrieveReferenceResult
                ERRORS { NavigationError, DSAError }
```

RetrieveReferenceArgument ::= DistinguishedName

RetrieveReferenceResult ::= Reference

It should be noted that, in most cases, a user will access the Directory via their home DSA and the Retrieve Reference operation will not require distributed navigation.

This section has described the distributed execution of single entry operations and the implementation of distributed access and integrity controls. The following section extends this work to consider multiple entry operations with more complex execution phases.

5.4.3. Multiple entry operations

This section considers the distributed execution of those directory operations accessing multiple entries. These operations are generally characterised by complex execution phases involving interaction between several DSAs. The operations considered are *List Subordinates* and *Search*.

The List Subordinates operation

The List Subordinates operation assumes the specified parent entry as the initial target for its navigation phase. In many cases, the execution phase accesses entries local to the coordinating DSA. This requires no additional DSA interaction beyond that needed for checking filters in Access Control Lists. However, there could be cases where children will be the responsibilities of many different DSAs. For example, the children of the entry */c=b/o=e/* in figure 5.3. The coordinating DSA could still determine the names of these children from its minimal knowledge. However, it would be unable to check their Access Control Lists for the necessary *detect* access category.

The necessity to check access controls for child entries means that the coordinating DSA will have to contact its immediately inferior DSAs to find the names of those child entries the user is permitted to know. Furthermore, these immediately inferior DSAs may have to check filters at the user's home DSA.

A worst case execution phase for the List Subordinates operation is shown in figure 5.10.

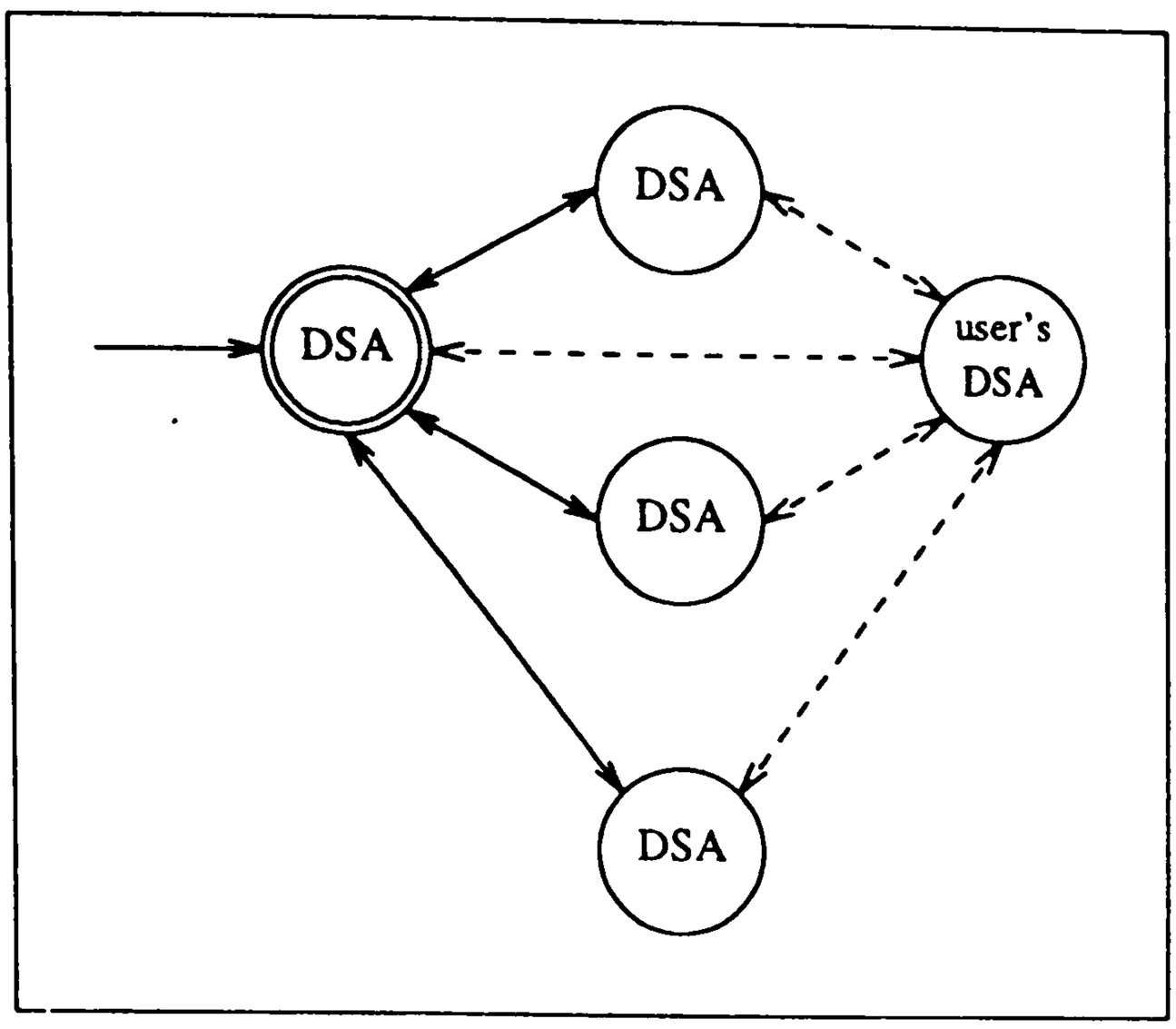


Figure 5.10: Worst case distributed execution
of the List Subordinates operation

This proceeds as follows:

- The coordinating DSA checks access rights for the parent entry. This may involve interaction with the user's home DSA.
- The coordinating DSA uses its minimal knowledge to derive the names of remote child entries and the references of their DSAs.
- The coordinating DSA requests a *Read Entry* operation on behalf of the user at each of these DSAs.
- The immediately subordinate DSAs may check Access Control Lists by interacting with the user's home DSA. They then return the results to the coordinating DSA.
- The coordinating DSA collects the results and returns them to the user.

It should be noted that only adjacent DSAs are involved in the execution phase of the List Subordinates operation and all DSA interactions therefore require a single hop. Furthermore, in the majority of cases, this operation is not likely to require interaction between the coordinating DSA and its immediately inferior DSAs.

The Search operation

The Search operation has a more complex version of the execution phase specified for the List Subordinates operation. The target entry for this operation is the root of the subtree to be searched and the coordinating DSA is therefore the DSA responsible for this entry. The major difference to the List Subordinates operation is that access might be required for all entries in a subtree. This may require recursive DSA interactions.

The execution phase of the Search operation involves the following steps:

- The coordinating DSA performs a local search and collects local results. This involves checking access controls.
- The coordinating DSA finds the references of all immediately inferior DSAs responsible for entries in the named subtree and whose fragments contain entries within the specified search depth.
- The coordinating DSA constructs new search operations with the relevant children as roots of subtrees and with the search depth suitably reduced. These operations are called at the relevant immediately inferior DSAs which recursively repeat these steps.
- The coordinating DSA collects the results from each inferior DSA and combines them with the local results. These are finally returned to the user.

Each DSA involved in the recursive search may have to contact the user's DSA to check access control lists. The distributed execution of the search operation is shown in figure 5.11.

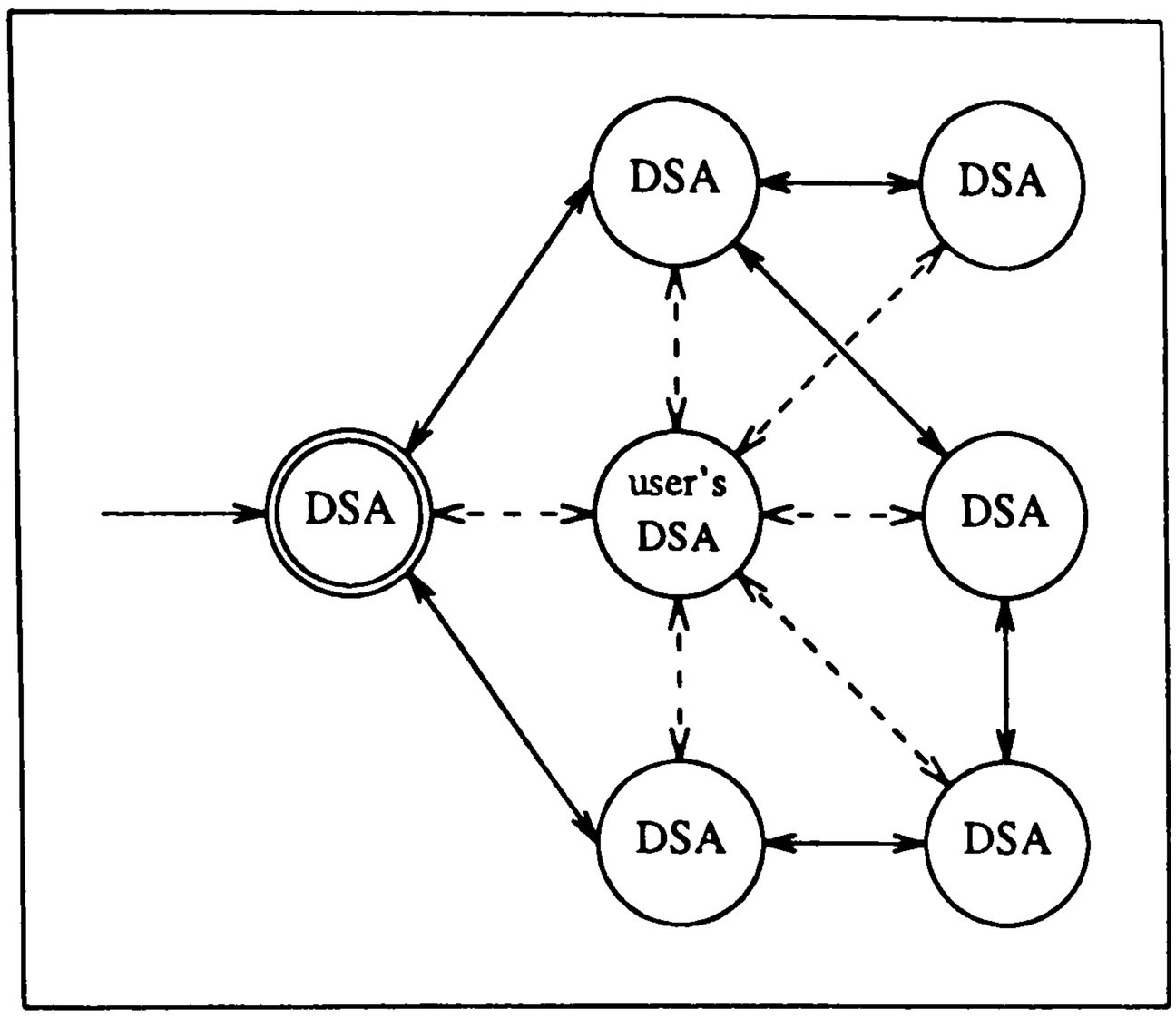


Figure 5.11: Distributed execution of the Search operation

Figure 5.11 shows that a DSA might be contacted more than once during a recursive search in order to access different fragments.

The important aspects of distributed searching are recursive DSA interactions and the suitable adjustment of search depths and subtree names.

In conclusion, the execution phases of multiple entry operations may require interaction between many DSAs. These interactions may even involve the recursive execution of operations. It should be noted that all of the above interactions occur between adjacent DSAs and therefore only require one navigation step thus reducing the navigation overhead. Furthermore, they only require the use of minimal knowledge.

The following section discuss the execution phases of those operations manipulating attribute and entry definitions.

5.4.4. Distributed management of entry and attribute definitions

Section 4.3 specified a number of operations to manipulate attribute and entry definitions. These are listed in the table below.

Operations manipulating entry and attribute definitions
Add Entry Def Delete Entry Def Suspend Entry Def Reinstate Entry Def
Add Attribute Def Delete Attribute Def Suspend Attribute Def Reinstate Attribute Def

This section discusses the distributed execution of these operations. This discussion must account for the downward replication of definitions as described above.

The target entry for each of these operations is the root of the subtree defining the scope of the affected definition. The coordinating DSA is therefore the DSA responsible for this entry. For example, an operation creating an attribute definition of type *group member* with scope */C=GB/O=Nott.Uni/* will be directed at the coordinating DSA responsible for the entry */C=GB/O=Nott.Uni/*. The following paragraphs examine the addition, deletion and suspension of definitions.

Adding definitions

The addition of a new entry definition may require that several existing entry and attribute definitions are validated for existence and compatibility. Due to the downward replication of definitions to all DSAs within their scopes, all the required definitions will be present within the coordinating DSA and validation will not require any DSA interactions. However, the converse problem of ensuring the downward replication of the new definition must be solved. This also applies to the addition of new attribute definitions.

For example, the addition of an entry definition of type *distribution list* with scope */C=GB/O=Nott.Uni/OU=CS/* might require validating the existence of the attribute definitions *member* and *administrator* with scope */C=GB/* and */C=GB/O=Nott.Uni/* respectively. Copies of both of these definitions would be present at the coordinating DSA (responsible for */C=GB/O=Nott.Uni/OU=CS/*) and therefore validation would be a local procedure.

The execution phases of the *Add Attribute Def* and *Add Entry Def* operations therefore proceed as follows.

- The coordinating DSA validates the existence and compatibility of other definitions referenced by the new definition. This does not require any DSA interactions.

- The coordinating DSA checks access controls. This may involve interaction with the user's DSA.
- The coordinating DSA adds the new definition to its local information base.
- The coordinating DSA propagates the new definition to its immediately inferior DSAs. These recursively propagate it to their immediately inferior DSAs thus ensuring the downward replication of the new definition.

The downward propagation of a new definition therefore occurs on a stepwise basis between adjacent DSAs. This may be achieved by the introduction of the *New Entry Def* and *New Attribute Def* DSP operations. The New Entry Def operation is specified by the following ASN.1 structures. The New Attribute Def is similar and is fully specified in Appendix A.

```
NewEntryDef ::= ABSTRACT-OPERATION
                ARGUMENT NewEntryDefArgument
                RESULT  NewEntryDefResult
                ERRORS { DSAError, NavigationError }

NewEntryDefArgument ::= EntryDefinition

NewEntryDefResult ::= NULL
```

Each DSA receiving one of these operations adds the specified entry or attribute definition to its information base and then calls the operation at its immediately inferior DSAs.

The distributed execution of operations adding new entry and attribute definitions is shown in figure 5.12 below

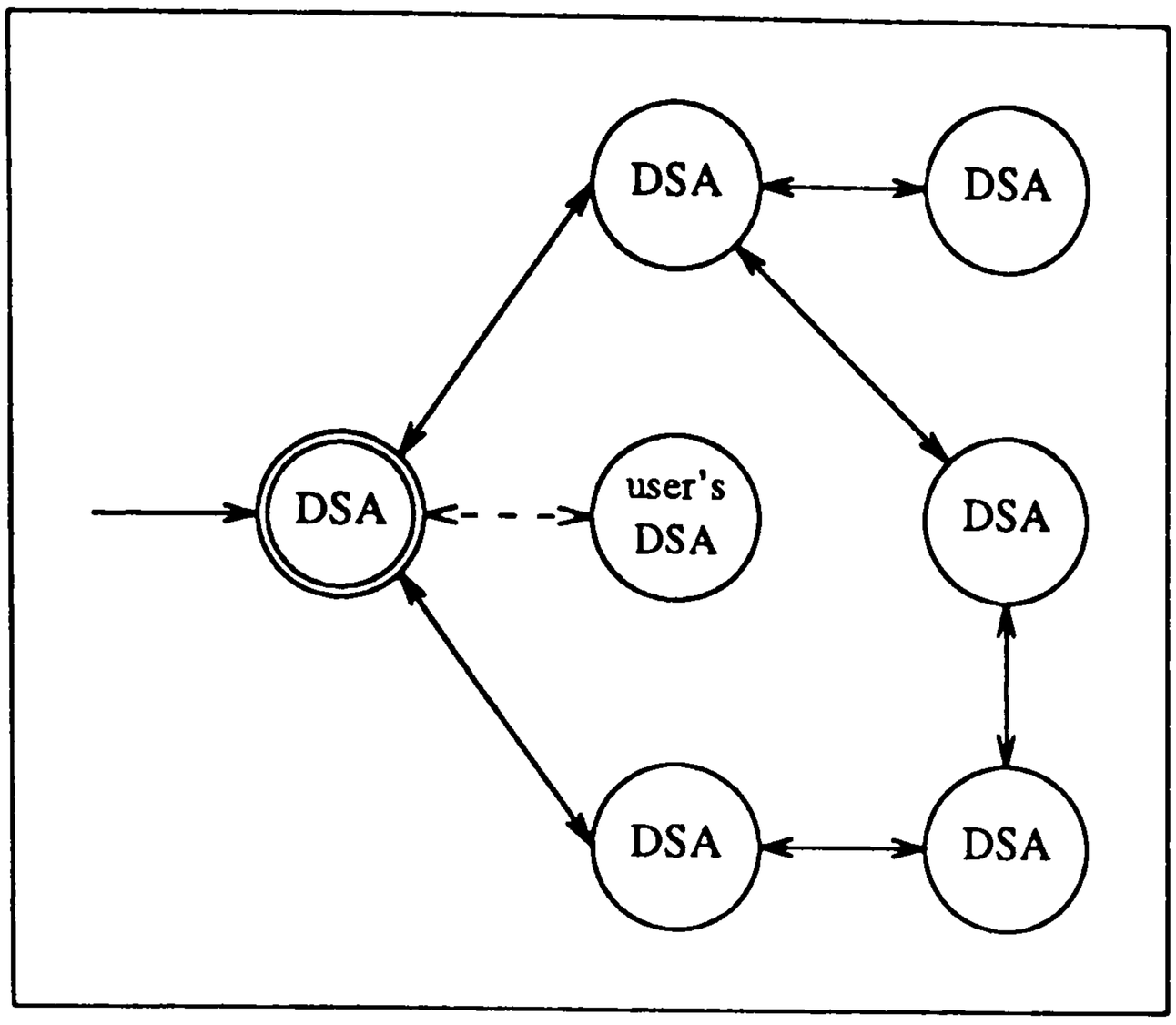


Figure 5.12: Distributed execution of the
Add Entry Def and Delete Entry Def operations.

It should be noted that access rights are only checked by the coordinating DSA. Once the coordinating DSA has added the new definition all inferior DSAs must accept the propagated update provided that it is from their immediately superior DSAs.

Deleting and suspending definitions

The deletion and suspension of entry and attribute definitions have similar execution phases to their addition. This involves the checking of constraints and access controls by the coordinating and user's home DSAs and then recursively propagating the alterations to all immediately inferior DSAs. Downward propagation of deletion and suspension requires the introduction of the new *Suspend Definition*, *Reinstate Definition* and *Delete Definition* DSP operations. These are fully specified in appendix A.

There are two additional problems with the deletion and suspension of definitions.

- The first is that a definition cannot be deleted while it is "still in use". A definition is defined as being *in use* if it is referenced by any other entry or attribute definitions or if it constrains any existing entries and attributes within the DIT. This condition requires that the use of a definition is checked by all DSAs within its scope before it can be deleted or suspended.

- The second concerns the time delay between checking the use of a definition and propagating its deletion or suspension. In the intervening time between these events, a new entry of attribute might be created referencing the definition. The definition would then be deleted while in use, thus creating an inconsistency.

Downward checking of the use of a definition is achieved by the *Check Definition Use* operation recursively acting between DSAs responsible for a DIT subtree. Given a type, scope and definition type it returns an indication of whether the definition is in use within the specified scope. This operation has the following ASN.1 specification:

```
CheckDefinitionUse ::= ABSTRACT-OPERATION
                ARGUMENT CheckDefinitionUseArgument
                RESULT  CheckDefinitionUseResult
                ERRORS { DSAError, NavigationError }

CheckDefinitionUseArgument ::= SET {
                type [0] PrintableString
                scope [1] Name,
                def-type [2] INTEGER { attribute(0), entry(1) } }

CheckDefinitionUseResult ::= BOOLEAN
```

The second problem arises due to the time delay between checking the use of a definition at a DSA and propagating the deletion of the definition to the DSA. The *Check Definition Use* operation might return an indication that the definition is not in use. However, a new attribute or entry for this definition might be created at an inferior DSA before the propagated deletion arrives. This would result in a definition being deleted while still in use.

The solution to this problem further highlights the advantage of allowing the suspension of information. Before deletion, a definition may be suspended disallowing its further use in creating new entries, attributes and definitions. After a suitable period of time the definition can then be deleted, safe in the knowledge that it has not been used to create new information. Suspending a definition before removal provides a sensible method of phasing out information providing that the time delay between suspension and deletion is suitably long.

The conclusion to this section is that the management of definitions requires the recursive downward propagation of updates to all DSAs within their scope. This is achieved by a number of new DSP operations. Suspension of definitions before deletion will ensure that conflicts do not occur due to time delays in the propagation of operations between DSAs.

5.4.5. Summary of distributed operations

The previous sections have discussed the distributed execution of operations within the Directory system. They have described the interactions between DSAs in terms of a number of new DSP operations. They have also considered distributed support for access and integrity controls. The following are the major conclusions of this work.

- Each distributed operation is executed in two phases. The *navigation phase* describes its navigation from the initial DSA to a *coordinating DSA* responsible for the target entry. The *execution phase* describes the interaction between a set of DSAs in order to perform the operation.
- The arguments and results of distributed operations include *envelopes* representing the navigation and error detection information specified in section 5.3.
- Checking of access controls may require interaction with a user's *home DSA* in order to match a filter. The efficiency of distributed access control is greatly increased by including *user information* within operation envelopes. This information specifies a user's name and the reference of their home DSA.
- The checking of integrity constraints does not require any DSA interaction providing that entry and attribute definitions are replicated to all DSAs within their scope.
- Most operations access a single target entry and consequently their execution phases require little DSA interaction.
- The execution phases of a few operations may require complex DSA interactions. However, these occur recursively within a DIT subtree and therefore only require single hop interactions between adjacent DSAs.
- Problems arising from the delays between checking the use of entry and attribute definitions and propagating their deletion to the necessary DSAs can be avoided by the use of the *Suspend Entry/Attribute Def* operations.

An important final point is that, due to the above mechanisms, all interactions between DSAs involved in the execution phases of operations only require one navigation step. This removes the need for costly navigation and therefore increases the efficiency of the distributed Directory. Furthermore, many instances of operations will not require any DSA interactions during their execution phases. For example, operations not matching filters or *List Subordinates* operations where the parent and child entries belong to the same DSA.

5.4.6. A comparison of chaining and referral

Section 5.1.1 described a number of modes of interaction between DSAs when navigating an operation during its navigation phase. These are *chaining*, *DSA referral* and *DUA referral*. This section compares these different modes and contrasts their strengths and weaknesses.

Chaining

Chaining involves the recursive invocation of an operation at successive DSAs in a navigation sequence until a responsible DSA is located. This DSA then coordinates the execution of the operation and returns the results along the chain to the user. Each DSA establishes and maintains a connection with the next DSA in the chain until it receives the results of the operation.

A major advantage of chaining is that it facilitates the automatic propagation and correction of knowledge. Each DSA in a chain is able to exchange references with its neighbours using the information present in operation and result envelopes.

A major disadvantage of chaining is that overall control of navigation is distributed between a number of DSAs. This makes it difficult for any one DSA to take control of navigation or trace the course of an operation. Furthermore, accounting may be difficult because the cost of establishing and maintaining connections will be borne by many different DSAs. Chaining within a directory running over a network such as British Telecom's *Packet Switching Service* (PSS) which charges for holding open connections, even without data transmission, might become expensive.

DSA referral

DSA referral involves the initial DSA assuming control of navigation. This DSA contacts a number of other DSAs each of which performs a navigation step and either returns (i) the results of the operation or (ii) a referral to a closer DSA.

A major advantage of DSA referral is that one DSA has overall control of the progress of the operation. This DSA may implement its own loop control if it wishes. Furthermore, this DSA contacts several other DSAs and therefore has a chance to learn new opportune knowledge and check its existing references. Accounting is also likely to be simpler where a single DSA initiates all of the connections for an operation.

A disadvantage of DSA referral is that the automatic correction of inconsistent knowledge is more difficult. The DSA performing the last navigation step and the DSA performing the next step (and therefore discovering the navigation error) are separated by the initial DSA controlling navigation. There is therefore no direct open connection between them which

can be used to support the *Update Reference* operation.

DUA referral

DUA referral is similar to DSA referral, except that it is the user's DUA which assumes overall control of routing.

This has a large advantage in that it is easy for the user to influence the course of the operation and even to terminate an operation in mid course if they wish. Furthermore, it gives DUA designers the freedom to implement their own control and navigation mechanisms with separate DUA knowledge management if required.

There are several major disadvantages with DUA referral. Firstly, it does not support the automatic correction of knowledge inconsistencies. There is no inter-DSA communication and hence no chance for automatic error detection and correction by the mechanisms specified in section 5.3.3. DUA referral also doesn't encourage DSAs to expand and check their opportune knowledge using the envelope information in DSP operations. In general, system management is the responsibility of the Directory system and should not be reproduced within each DUA.

Secondly, DUA referral requires that the user bind to a different DSA with each successive navigation step. This may place a greater authentication overhead on the Directory and introduce delays as the user's home DSA is located by each new DSA.

Summary

It is difficult to analyse the relative expense of the different modes of directory operation due to the large number of factors affecting the cost of operations (e.g. volume of information, connection time, different charging algorithms etc.). However, given the distributed model specified by this chapter, the following general observations can be made.

- *DUA referral* is a generally inefficient and unsuitable mode of operation. It introduces aspects of directory operation such as knowledge management and navigation into DUAs although they logically belong within the Directory system. However, DUA referral is worth supporting because it offers users and DUA designers the chance to implement their own non-standard methods of interaction if they really wish.
- Both *chaining* and *DSA referral* have their advantages and may be used under different circumstances according to user and DSA policy. These are recommended as the normal modes of directory operation.

This thesis therefore proposes that the Directory should support all three modes of directory operation with chaining and DSA referral recommended for normal use. DSAs might

choose to support different local policies and users might also be given a choice of service they wished to use.

5.5. Summary of knowledge, navigation and distributed operations

This chapter has described the directory local conceptual model in terms of the distribution of the directory global conceptual model between a number of Directory System Agents. This work has included partitioning the Directory Information Tree between DSAs and describing the distributed navigation of operations. The DSA interactions required to execute each directory operation have also been specified.

The major goals of this chapter were to specify the distribution of the Directory while supporting DSA autonomy and, as far as possible, allowing the automatic management of the Directory system itself. The following sections present the major conclusions of this work.

Partitioning the DIT and navigation

The DIT can be divided into *fragments* where each fragment is the responsibility of a single DSA. Each DSA may be responsible for several fragments and these responsibilities are represented by the DSA's *reference*.

An operation is navigated from an initial DSA to a target DSA via a sequence of DSAs, each of which performs a *navigation step* moving the operation one step closer to the target. In order to perform a navigation step, a DSA must have *knowledge* of other DSAs responsibilities. Each DSA possesses *minimal* knowledge containing the references of all *adjacent* DSAs. Minimal knowledge determines the connectivity between all DSAs and guarantees that navigation can always occur. In addition to its minimal knowledge, a DSA may use *opportune* and *replicated* knowledge to optimise the navigation process and to increase the robustness of the Directory service. Opportune knowledge may be acquired by a number of ad-hoc methods such as caching.

The management of knowledge

This chapter also addressed the problem of managing knowledge as the configuration of the Directory system alters.

Minimal knowledge is essential for correct navigation and must be maintained in a consistent state by a well defined protocol. This is possible due to the small number of well known DSAs whose minimal knowledge is affected by a change to another DSA's reference.

On the other hand, the consistent update of opportune knowledge is virtually impossible due to its large volume and acquisition by ad-hoc methods. Instead, the navigation procedure must detect and accommodate inconsistencies in opportune knowledge. It should also automatically correct them where possible.

Knowledge inconsistencies may be detected by the *hop count*, *intended reference* and *route checking* procedures within each DSA. Automatic corrections may be implemented via the *Update Reference* operation between two DSAs.

In general, DSAs may swap references during navigation therefore improving and maintaining their opportune knowledge.

Execution of distributed operations

The execution of distributed operations within the Directory system may involve a number of DSAs interacting in quite complex ways. This chapter specified the distributed execution of operations so that the number of DSA interactions was minimised thus reducing communication costs and time delays.

The execution of each operation can be viewed in terms of a standard *navigation phase* followed by an *execution phase*. The execution phase of an operation may require support for distributed access and integrity control. The efficiency of these controls may be increased by the following two mechanisms:

- Including the user's name and the reference of their home DSA in each operation envelope in order to facilitate the matching of filters against the user's entry.
- The downward replication of entry and attribute definitions to all DSAs within their scope in order to confine the validation of integrity constraints to the local DSA.

More complex operations such as *Search* may require recursive interactions between DSAs. However, these involve contact between adjacent DSAs and do not require complex navigation during their execution phases. In addition, new operations for recursively propagating changes to entry and attribute definitions have also been specified.

In general, the directory should use the chaining and DSA referral modes of operation. However, in exceptional circumstances, the DUA referral mode might also be adopted.

This concludes the summary of the basic distributed Directory model. Chapter 6 extends this model to support replicated information thus increasing the efficiency and robustness of distributed operation.

Chapter 6

A Directory Replication Model

This chapter extends the local conceptual model of chapter 5 to support the *replication* of information between Directory System Agents. This work aims to develop an abstract model for replication within the Directory service. This high-level model provides a general framework for the management and update of replicated information and could be used as the basis for different Directory implementations, adopting their own specific replication mechanisms.

The approach of this chapter is to examine replication within general distributed databases and then to develop a directory replication model based on a combination of current approaches.

Replication is a large and complex subject and this chapter does not aim to cover the full range of replication issues in detail. In particular, it does not describe the specific, detailed implementation of a directory replication mechanism. However, the proposed model addresses many important replication issues at an abstract level thus providing the foundations for future work in this area.

In general, the following directory replication model aims to increase both the efficiency and robustness of directory operation. A major requirement of this work is support for the consistent update of information between a loosely coupled set of autonomous DSAs. The model also describes the structure of *replicated* knowledge and examines the impact of replication on the work of chapter 5.

Management issues include the management of replicated information and the use of *replication policy statements* to describe replicated information in a flexible manner. The model also considers the management of *replication agreements* between DSAs and specifies the basic support required for establishing and terminating replication in terms of a number of new DSP abstract operations.

- Section 6.1 provides an overview of replication issues. It explains the motivation for replication within general databases and, more specifically, within the Directory service. This section then outlines the major issues covered by this chapter.
- Section 6.2 examines how the issues of update propagation and consistency have been addressed by distributed database theory.

- Section 6.3 proposes a general directory replication model based on the results of section 6.2. This model involves tightly coupled *DSA clusters* supporting loosely coupled inter-cluster replication.
- Section 6.4 examines replication within DSA clusters.
- Section 6.5 examines replication between DSA clusters including the use of replication policy statements. This section also specifies the structure of replicated knowledge and investigates the management of replication agreements.
- Section 6.6 considers the issue of crash recovery within the Directory service.
- Section 6.7 summarises the impact of replication on the navigation and execution of directory operations as discussed by chapter 5.

Finally, section 6.8 summarises the work of this chapter.

6.1. An overview of replication issues

The following sections explain the motivation for supporting replication within distributed databases and indicate possible uses for replication within the Directory service. This is followed by a discussion of the major problems to be solved by this chapter.

6.1.1. Why replicate within distributed databases?

Replication of information means that a given piece of information may be physically stored or copied at one or more nodes of a distributed database [DATE83]. In directory terms, this means that a piece of data may be stored within one or more DSAs simultaneously. Replication should be supported within distributed databases for the following reasons [LAR85]:

- It may improve performance due to a reduction in inter-node communication. For example, a node may be able to use a local copy of information as opposed to contacting a remote node.
- It may improve the *robustness* of the system by increasing the *availability* of information. For example, if a node responsible for some information is unobtainable, another copy at another node may be used.

These gains in performance and robustness are made at the expense of increased storage overheads and system complexity and there is a general tradeoff between the advantages and disadvantages of replication. In the case of the Directory service where DSAs are physically widely separated, communication costs will be high, both in terms of time and money, and it is likely that replication will improve performance.

6.1.2. Example uses of replication within the Directory service

The following paragraphs briefly describe three possible uses of replication specific to the Directory service.

- Navigation and name verification are fundamental to the operation of the Directory service. Replication of the higher portions of the DIT within lower DSAs would therefore enhance directory performance by reducing the number of DSA interactions required during navigation. Furthermore, entries high in the DIT are likely to be updated infrequently and so the maintenance overhead would be relatively small. This use of replication also increases the robustness of the Directory.
- Chapter 5 discussed the downward replication of entry and attribute definitions to all DSAs within their scopes. This use of replication greatly reduces the communication costs of operations such as *Add Entry* which frequently reference definitions higher in the DIT. Once again, it is expected that definitions will be relatively stable thus minimising the maintenance overhead.
- DSAs could replicate copies of entries which are immediately subordinate to their fragments. This would reduce the number of DSA interactions required during the *List Subordinates* operation.

The above are just a few examples of the many possible uses of replication to increase the robustness and performance of the Directory service. In general, DSAs should replicate information which is frequently accessed but which is relatively stable. Conversely, optimising the use of rapidly changing information might best be achieved via the use of opportune knowledge as described in chapter 5 and not via replication.

The directory replication model should support the ability to define general replication strategies beyond those described in the preceding paragraphs and users/administrators should be able to specify *replication policies* utilising replication in a general manner.

The remainder of this section describes the major issues to be addressed when specifying the directory replication model.

6.1.3. General replication issues

This section outlines some of the issues which have proved relevant to replication within general distributed databases and which should be addressed by this chapter. These may be summarised by the phrases *data fragmentation* [NAV85], *query processing* [CHU82], *replication transparency* [DATE83] and *concurrency control* [BERN81]. The following paragraphs describe these issues in terms of the relational data model. However, they are generally applicable to other data models, including that of the Directory service.

Data fragmentation

A distributed database supporting replication partitions data into *replicated fragments*, each of which may be stored at one or more database nodes. The question arises as to what is a suitable unit of fragmentation. The general solution for relational systems defines a fragment as being any sub-relation derived from a base relation via an arbitrary combination of *projection* and *restriction* operations.

This chapter must define suitable replicated fragments for the directory model in terms of the DIT, entries and attributes. Replicated fragments need not be the same as the partitioned fragments specified by chapter 5 (section 5.2).

Query processing

Query processing describes the method by which a user specified query is broken down into sub-queries which may be performed at different nodes of a distributed database. Query processing within distributed relational systems, supporting replication, is a complex business. Queries may suggest a large number of possible execution strategies resulting in widely varying performances (one simple query given by Rothnie and Goodman [ROT77] predicts execution times varying between 1 second and 2.3 days depending on which strategy is chosen!). Consequently, much effort has been devoted to the problem of query processing and many distributed databases support dynamic query processors which choose a suitable policy at run time [EPS78, HEV82].

Directory query processing was discussed in section 5.4 and is generally far simpler than for relational systems due to the less complex structure of directory operations compared to arbitrary relational queries. However, this chapter must consider the effect of replicated information on the distributed execution of directory operations. The impact of replication on navigation and name verification will be of particular interest.

Replication transparency

Chapter 2 described the need for the Directory to support replication transparency at the external and global conceptual levels. Details of locating and maintaining fragments should therefore be handled by the system, not the user, and, in general, the user should view information as if only one copy existed. The joint effect of *location transparency* and *replication transparency* are summarised by the following sentence: *A distributed system should look like a centralised system to the user*.

On the whole, this is true of the Directory and this chapter ensures that the details of replication are confined to the local conceptual layer thus preserving transparency. However, there

may be exceptions such as *transient inconsistency* described below.

Concurrency control

The major problem to be solved by replication models is that updates to one copy of information must be propagated to all copies in a consistent manner. Fundamental to this problem is the concept of the *transaction*, generally defined as a database action which is "atomic" and therefore either entirely succeeds or fails [LAR85]. A single transaction, from the user point of view, may involve several distinct actions on replicated copies of information at different nodes and, if transaction atomicity is to be preserved, these must be coordinated to leave the database in a consistent state in spite of possible system failures. This is referred to as *concurrency control* and involves *update propagation*.

Relational databases often allow user specified transaction control therefore increasing the difficulty of maintaining consistency whereas the Directory defines a transaction as a single directory operation. This is a simpler proposition. However, the problem of consistency between replicated information still has to be addressed and, in order to do this, it is worth considering general database approaches to updating replicated information. This is the task of section 6.2.

6.1.4. Replication management issues

There are two management issues to be addressed by this chapter.

The first concerns the management of replication as new information is added to the Directory. The Directory should be able to determine whether new information is covered by existing replication agreements without consulting human administrators. This requires support for *replication policies* where administrators specify the types of information they wish to replicate in generic terms. The system then uses these policies to manage the details of precisely what information is replicated.

The second concerns the management of replication agreements themselves. The Directory should support mechanisms for dynamically establishing and terminating agreements between DSAs. These replication agreements should not be managed by an extra-directory mechanism.

This concludes the overview of the replication issues to be addressed by this chapter. The following section explores the important issue of update propagation in more detail and explains how consistency is achieved within general distributed database systems. This work is then applied to the Directory system.

6.2. Update propagation and consistency

The problem of maintaining transaction atomicity when applying updates to multiple copies of information is one of *update propagation* and is fundamental to the operation of replication mechanisms. This section describes general approaches to update propagation which have emerged from work on distributed databases.

The following two examples clearly indicate the difficulties of consistently updating replicated copies of information.

Example 1: Conflicting writes

The first example demonstrates that simultaneous updates to different copies of information can result in inconsistency. Figure 6.1 shows two nodes of a distributed database, both maintaining copies of information A.

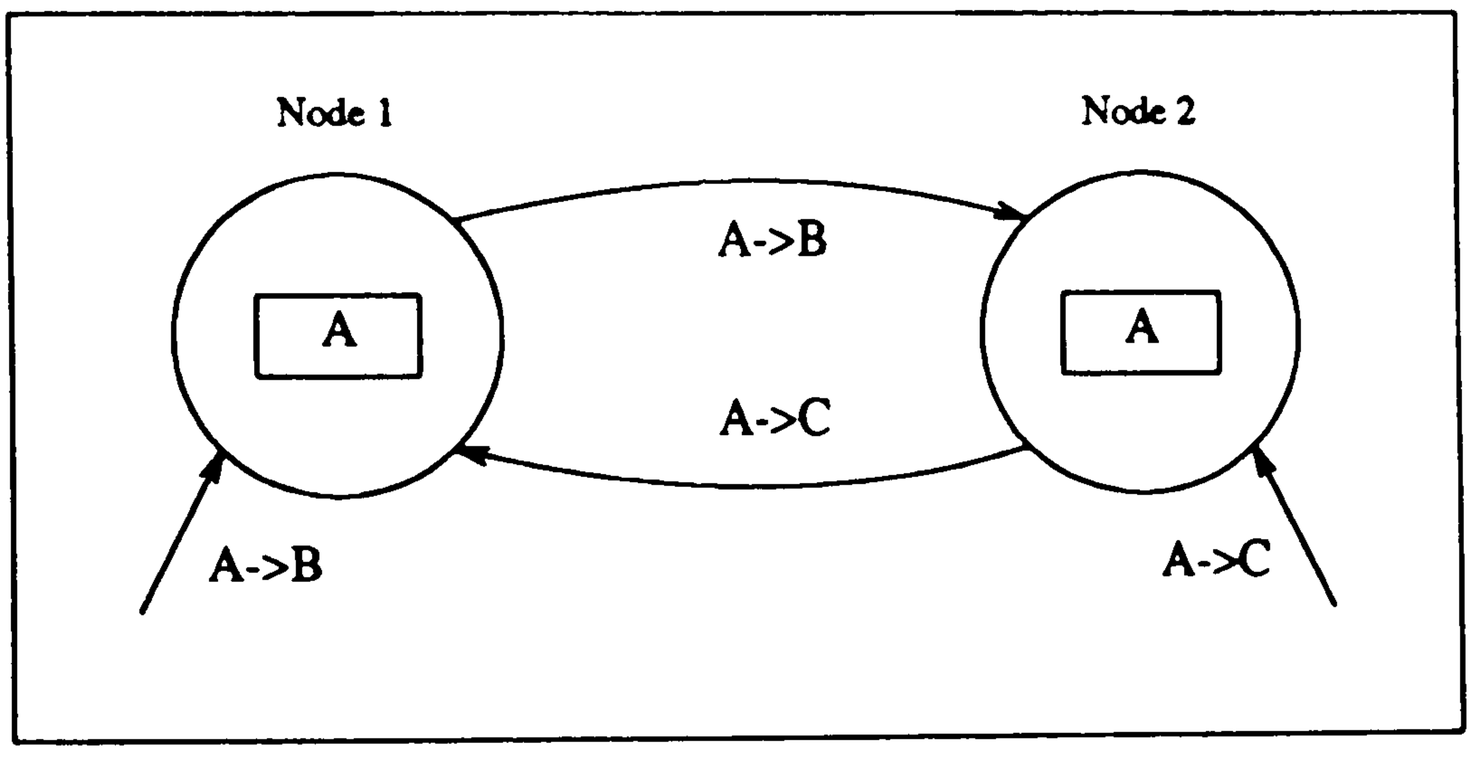


Figure 6.1: Two conflicting writes

Node 1 receives an operation updating *A -> B* and, at approximately the same time, node 2 receives an operation updating *A -> C*. Consequently, nodes 1 and 2 update their local versions of A and then attempt to mutually update each other. Both updates are not logically possible because the information A no longer exists and they therefore fail leaving inconsistent copies of information. This example involves two nodes and a simple operation. The situation can become far more complex involving many nodes and many updates including the addition and deletion of information. In general, this type of conflict is referred to as a *lost update* problem [BERN81].

Example 2: Conflicting read and write

A more subtle form of conflict occurs where the update of information disturbs a read which is critical to some other update operation. Figure 6.2 shows two nodes holding copies of information A and D.

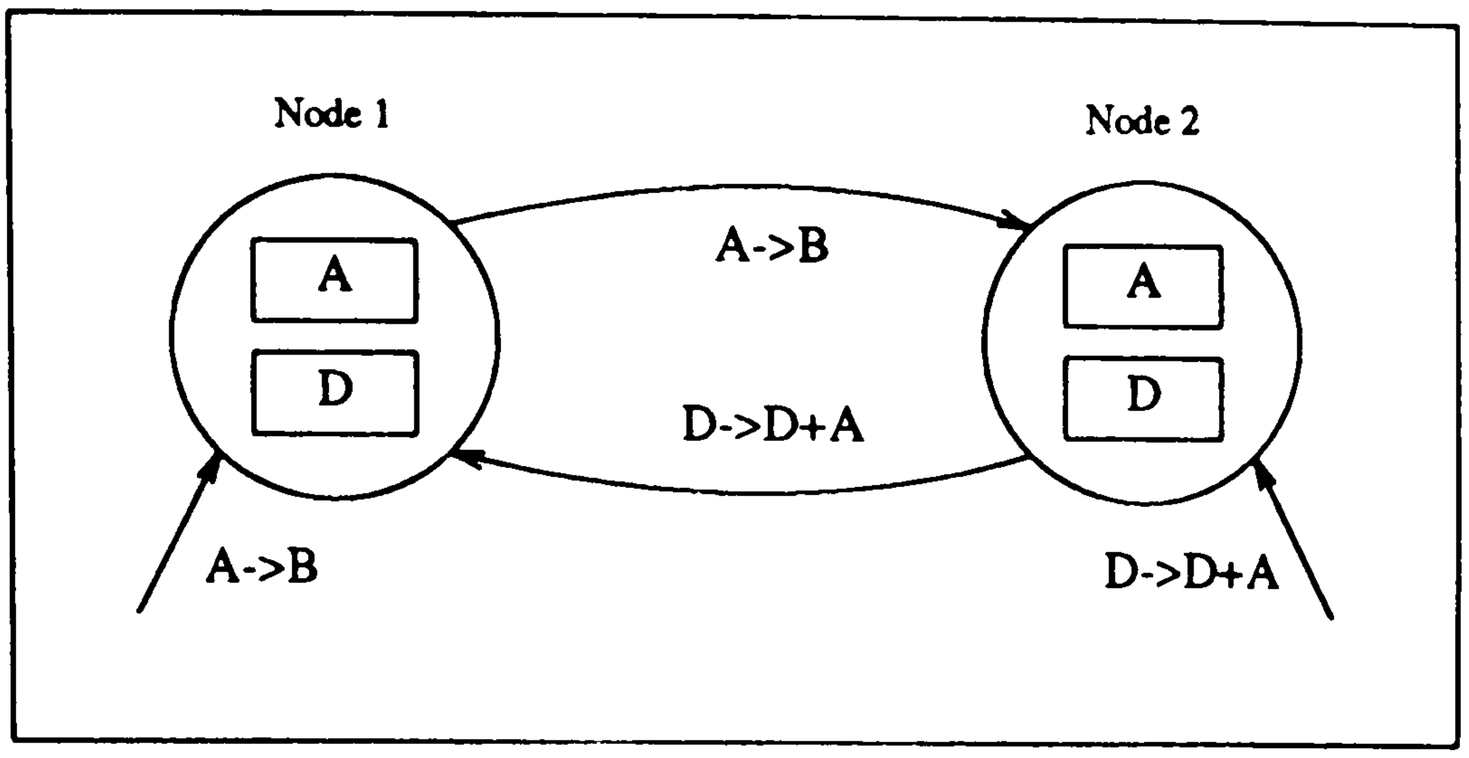


Figure 6.2: Conflicting read and write

Node 1 receives an operation updating A -> B and node 2 receives an operation updating D -> D + A. The update to A will occur at both nodes. However, the update to D will use different values of A yielding the values D + B at node 1 and D + A at node 2.

These are just two examples of update propagation problems. In general, any transaction updating the database has a *readset*, which is the set of information which must be read or checked in order to complete, and a *writeset*, which is the set of information which is updated by the transaction. Two transactions may conflict if the readset or writeset of one overlaps with the writeset of another. Example 1 showed conflict due to overlapping writesets and example 2 showed conflict due to a writeset overlapping a readset.

The following sections describe several approaches to preventing inconsistency due to conflicting updates. A more detailed discussion of these mechanisms can be found in [KOH81] and [BERN82].

6.2.1. Single master update

The *single master* update policy solves the conflicting update problem by distinguishing between copies of information used for reading and copies used for writing. For each replicated fragment of information, there is one *master* copy which is writeable and readable and

a number of *slave* copies which are read-only. Slave copies may be used for satisfying pure read requests but any update requests must be directed at the master copy. Furthermore, read requests belonging to the readset of an updating transaction must also be directed at the master copy. The imposition of single mastership excludes the possibility of conflicting updates initiating at different copies of information.

Update of information proceeds in the following way: An update request is directed at the node holding the master copy (the *master node*) and this node determines whether the update should succeed or fail. If the update succeeds, the master must then propagate the update to all slave copies (held at *slave nodes*). The main problem to be solved with this approach is a mechanism for propagating updates from the master to the slaves.

In general, the master node must maintain a list of all current slave nodes for each fragment and must ensure that each slave is updated. This is referred to as *master initiated update*. It is possible that a slave node may be unobtainable at the time of update and that, as a result, the slave copy will be inconsistent. The solution to this problem is one of *crash recovery* and may involve the master node queuing the update until it can be performed, the slave attempting to resynchronise with the master when the connection is re-established or a combination of these. Crash recovery is discussed more fully in section 6.7.

The advantages of the single master update policy are:

- Eventual consistency is ensured for any one transaction
- The protocol is relatively simple to manage and implement
- Updates require a minimum of communication between sites.

The disadvantages are:

- The concurrency of the system is drastically reduced by having a single master node. In some cases the master node may even form a bottleneck.
- Information may show *transient inconsistency* meaning that there will be a period of time, between the update being committed at the master and at all the slaves, during which copies will disagree. Transient inconsistency is guaranteed to be resolved eventually.
- The robustness of the system is reduced because of the dependence on the master node for updates. This can be expressed by saying that the *availability* of the information for update is limited. On the other hand, the availability of information for reading is increased.

A possible solution to the latter problem is to adopt a *shifting master* policy where slave nodes elect a new master on the event of the old master node becoming unobtainable.

However, this requires a great deal of coordination and introduces a large communication overhead between sites.

6.2.2. Snapshots

The *snapshot* replication mechanism [ADI80] is a variant of the single master policy which relies on slave initiated update rather than master initiated update. Master and slave copies are defined as before and updating operations must still be directed at the master copy. However, once the master has committed the update, it does not propagate it to the slave copies. Instead, slave copies periodically refresh themselves from the master.

Snapshots can be viewed as formalised *caches* and result in longer transient inconsistency. The period of time taken to restore consistency across all copies following an update may be considerably greater than for master initiated updates. However, this level of consistency may be satisfactory for many applications accessing the Directory, particularly when the rate of updates is low. The main advantage of snapshots is that they provide an extremely simple update propagation mechanism.

Both the single master and snapshot replication policies are *loosely coupled* mechanisms and are applicable to distributed systems consisting of loose *federations* of autonomous nodes.

6.2.3. Multiple master update

The problems with availability and robustness associated with single master update policies can be reduced by introducing the concept of *multiple mastership* where updates may be directed at more than one copy of replicated information. Multiple mastership means that more than one node may act as the master node for information and hence updates may originate from several nodes throughout the database. Consistency issues, as described in examples 1 and 2 above, present the main problems for multiple mastership and much effort has been devoted to finding mechanisms for preserving consistency in such an environment.

A mechanism enabling concurrent updates originating from different nodes, whilst maintaining consistency, must take a set of transactions, interleaved with respect to time, and leave the database in a state which would follow if they had executed in some serial order. Two different approaches to multiple mastership, based on *locking* and *timestamping*, are outlined by the following sections. Both of these can be thought of as *tightly coupled* solutions where updates are committed simultaneously at all replicated copies thus eliminating *transient inconsistency*.

Concurrency control by locking

The *locking* concurrency control mechanism requires that a transaction acquires locks on all copies of information required for completion thus blocking other transactions from interfering with it. Once the transaction has locked all the information it requires, it can proceed and, if successful, can commit updates at all copies and then release its locks. Each transaction must lock all information in both its readset and its writeset. To be specific, the following rules must be obeyed:

- The transaction must acquire a shared lock on at least one copy of all information to be read.
- The transaction must acquire an exclusive lock on every copy of information to be updated.
- A lock cannot be released until a COMMIT or ABORT.

The locking mechanism must determine a suitable course of action when a copy is unavailable due to the failure of a node. Coping with node failures may require the retransmission of messages and various crash recovery mechanisms.

There are two major disadvantages to distributed locking. The first is the large communication overhead required between nodes to acquire and release locks. Locking between n nodes requires the following transfer of messages:

n lock requests
n lock grants
n update messages
n acknowledgements
n unlock requests
(total of 5n messages)

This may be a large number of messages.

The second major disadvantage is the possibility of *global deadlock* which can occur when a number of transactions are waiting to acquire mutually held locks [BERN81]. Deadlock detection in non-distributed databases can be achieved by searching for *cycles* in lock request graphs. However, this is not possible in the distributed case because no single node holds enough information to construct such a dependency graph. In fact, effective deadlock detection within distributed databases presents an extremely difficult problem. Deadlock may be avoided by means of a centralised lock manager at a single node responsible for administrating locks but this results in a loss of nodal autonomy. Deadlock may also be prevented by a *wait-die* policy, terminating a transaction whenever it is unable to immediately obtain a required lock [ROS78]. This might become expensive due to many unnecessarily restarted transactions which would not have conflicted.

Although locking mechanisms are frequently implemented in centralised database systems, they seem unsuitable for providing distributed concurrency control due to the high communications overhead and possibility of global deadlock. The next section examines a different concurrency control mechanism based on timestamping information.

Concurrency control by timestamping

An alternative approach to distributed concurrency control is provided by *timestamping*. An overview of timestamping mechanisms can be found in [DATE83] and [BERN81]. Timestamping requires that each transaction is stamped with a globally unique time which can be compared with the times of other transactions so that they can be executed in some serial order. Concurrency control using timestamping operates in the following way:

- Each transaction is assigned a globally unique timestamp.
- Updates are not applied to copies until a successful end of transaction.
- Every piece of information carries the timestamp of the last updating transaction.
- If transaction T1 requests an operation conflicting with a younger transaction, T2, then T1 is restarted. Conflict occurs when T1 requests a read and T2 has already updated the information or T1 requests a write and T2 has already read or updated the information.

Two problems must be solved if timestamping is to be used as a method of providing concurrency control. Firstly, timestamps must be globally unique. This can be assured by structuring a timestamp as a combination of a clock reading and globally unique *node id*. Secondly, clocks running at different nodes must be synchronised, otherwise a node can effectively be shut out from updating information. Synchronisation of distributed clocks can be a tricky problem and there are a number of possible solutions. One solution states that clocks can be loosely synchronised by obeying the following rules [LAM78]:

- A node must increment its clock between every pair of consecutive events at the node.
- If a node with current clock reading t0 receives a message timestamped t1 then the clock is reset to max(t0,t1).

Timestamping overcomes the problems of global deadlock and high communication costs associated with locking. However, the policy of restarting transactions whenever conflict occurs may become expensive. Mechanisms based on *conservative timestamping*, *transaction classes* and *conflict graph analysis* can be used to increase the performance of timestamp based concurrency control. An overview of these mechanisms is given within [DATE83].

Summary of update propagation and concurrency control mechanisms

The major problem to be solved by replication mechanisms is that of ensuring the consistent update of different copies of information. Each transaction updating information has a readset and a writeset and inconsistencies may occur if the writeset of one transaction overlaps with the readset or writeset of another.

Distributed database theory has developed a number of mechanisms facilitating the consistent update of replicated information. The *single master* and *snapshot* mechanisms are *loosely coupled* solutions designating a single node as the master for each piece of information. Updates may only originate at the master node thus removing the possibility of conflicts. The *locking* and *timestamping* mechanisms are *tightly coupled* solutions allowing multiple masters for information and therefore supporting concurrent updates. Locking is an extension of the locking techniques often applied to centralised databases but is less suited to the distributed case due to a high communications overhead and the difficulty of detecting and managing global deadlock. Timestamping avoids deadlock problems and reduces the communications cost although mechanisms for synchronising clocks must be provided.

The following section begins the specification of a directory replication model utilising some of the techniques described above.

6.3. Overview of the directory replication model

This chapter proposes a model for replication within the Directory service utilising a combination of the above approaches. This section provides an overview of this model and therefore defines a framework for subsequent discussion.

The following characteristics of the Directory service affect the design of the replication model.

1 The rate of updates to directory information is generally expected to be far lower than the rate of reads.

2 Transient inconsistency is allowable for most directory applications. For example, contrast the case where the Directory provides an incorrect address and users are temporarily inconvenienced against a banking system which calculates interest incorrectly due to inconsistency. The former appears to be an inconvenience, the latter a disaster.

3 Directory information will usually be maintained on an organisational basis. This means that most updates for a piece of information will be directed at a limited set of *local* DSAs.

4 Directory services will operate across wide area networks (WAN). At the present time this means that long distance communication may be relatively slow although we can expect to see improvements in the future with the introduction of *high speed wide area networks (WAN)*.

Point 1 suggests that the loss of concurrency due to the adoption of a single master or even snapshot replication policy may be generally acceptable. The use of a single master policy is also justified by point 2.

Points 3 and 4 suggest that any multiple master policies should be constrained to operate between small sets of DSAs, probably belonging to the same organisation, and operating in a close environment such as on the same Local Area Network. Point 3 indicates that this would be sufficient for the majority of updates and point 4 suggests that it will be difficult to tightly couple wider sets of DSAs in the immediate future.

In conclusion, both the loosely coupled single master and snapshot update mechanisms seem generally applicable to the Directory service. However, in order to increase the robustness of the system, some limited use of tightly coupled multiple mastership may be useful. This chapter proposes that the Directory service should support a combination of single master, snapshot and limited multiple master replication strategies.

The proposed replication model views the Directory system as islands of tightly coupled DSAs called *DSA clusters*. The islands themselves may be interconnected in a loosely coupled manner. DSA clusters are therefore defined as groups of DSAs such that:

- The DSAs within a cluster are masters for the same fragments and are tightly bound by a multiple master update policy (tight coupling).
- Information may be replicated between DSAs from different clusters under a single master update policy (loose coupling).

The clustering of DSAs is shown in figure 6.3.

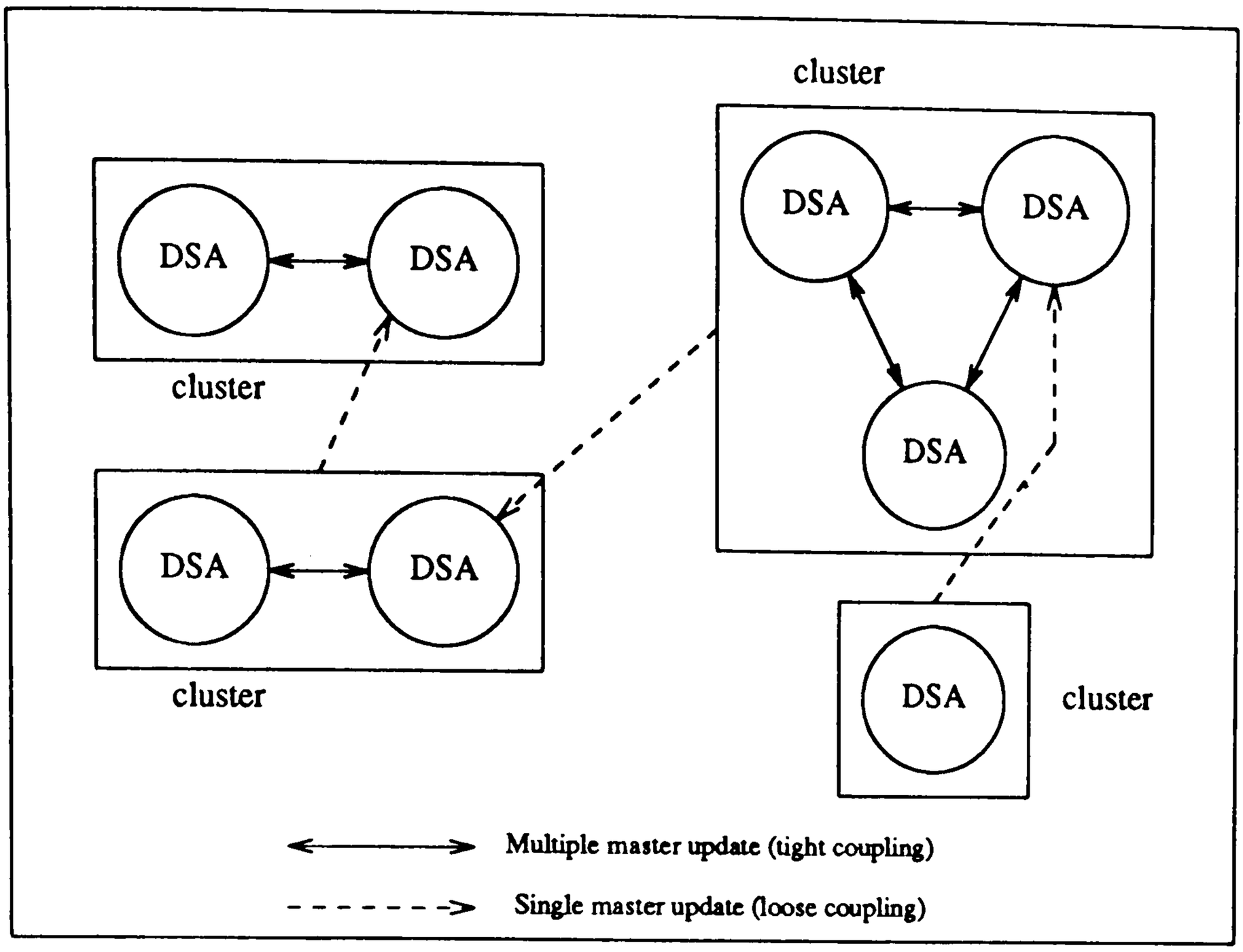


Figure 6.3: Directory replication using DSA clusters

The respective uses of tightly coupled replication within clusters and loosely coupled replication between clusters reflects the different motivations for replicating information.

- Replication between clusters improves the performance of the Directory system by allowing local DSAs to store copies of remote entries to be used during retrieval operations. It also defines replicated knowledge thus improving navigation.
- Replication within a cluster enhances the robustness of the Directory system by increasing the availability of information for update.

It follows that an individual DSA may hold master copies of some information and slave copies of other information.

It is important to understand that DSAs within a cluster are masters for identical fragments and that, in effect, the previous partitioning of the DIT between single DSAs (see section 5.2) has been replaced by a partitioning between DSA clusters.

Replication between clusters is supported by a single master update policy where the master node is really a DSA cluster. Thus, DSAs from a cluster act in unison as masters but independently as slaves.

The following sections examine different aspects of the directory replication model in greater detail. They discuss fragmentation, update propagation and the effects of replication on name verification and the distributed execution of operations. They also consider the management of replicated information and of replication agreements themselves.

Section 6.4 describes the replication of master fragments within a DSA cluster under a multiple master update policy. Section 6.5 discusses the replication of information between clusters using single master or snapshot update policies.

6.4. DSA clusters and multiple mastership

A DSA cluster is defined as a set of DSAs holding master copies of the same DIT fragments. This means that each DSA can only belong to one cluster and that each fragment can only have one master cluster.

DSA clusters increase the robustness of the Directory service by allowing multiple master update on a limited scale and must therefore be able to support a multiple master update policy without a crippling overhead. It follows that clusters will typically contain a small number of DSAs connected via good communication links. For example, an organisation might support a cluster of two DSAs responsible for its information and located on the same Local Area Network. The future introduction of high speed WAN may relax these physical constraints on clustering and allow larger and wider clusters.

The choice of multiple master replication policy varies from cluster to cluster depending on local conditions and DSA implementations. Thus, the directory replication model supports many different multiple master concurrency control mechanisms with each cluster choosing the optimum local solution. For example, one cluster of DSAs might be implemented as nodes of an existing local area distributed database supporting its own lower level update mechanism (internal layer) whereas a second cluster might support an explicit timestamp based protocol within the local conceptual layer. However, although the model allows many possible update mechanisms, this thesis notes that a timestamp based policy would seem to offer the best general solution.

The detailed specification of possible multiple master concurrency control mechanisms can be derived from other work on general distributed databases and is not addressed by this research. However, the following points mention some basic requirements of whichever mechanisms are chosen.

The following information must be shared between all DSAs in a cluster and must therefore be maintained in a consistent state by the multiple master update mechanism.

- All fragments belonging to the cluster.
- The references of all other DSAs in the cluster. This allows the multiple master update mechanism to determine the names of all master DSAs for a fragment.
- Minimal knowledge for all DSAs in the cluster. This guarantees successful navigation from any DSA within the cluster.
- Copies of all *replication agreements* with DSAs outside the cluster (see section 6.5 below). This supports the single master update mechanism.

The following information will remain specific to a single DSA and need not be shared with other DSAs in its cluster:

- Opportune knowledge.
- Any slave copies of information held by other clusters.

The introduction of DSA clusters has a large impact on the distributed directory model specified by chapter 5. In particular, replication within a DSA cluster affects the following aspects of the Directory's distributed operation.

- The structure of minimal and opportune knowledge.
- The execution of those operations updating directory information.

These issues are discussed below.

6.4.1. Revised structure of minimal and opportune knowledge

Previously, each DSA's minimal knowledge contained references to the DSAs responsible for immediately superior and inferior fragments of the DIT. Fragments are now the responsibility of a DSA cluster and each DSA's minimal knowledge should be extended to include references for ALL DSAs in the clusters responsible for immediately superior and inferior fragments. These clusters are, predictably, called the immediately superior and inferior clusters. The revised structure of minimal knowledge is shown in figure 6.4.

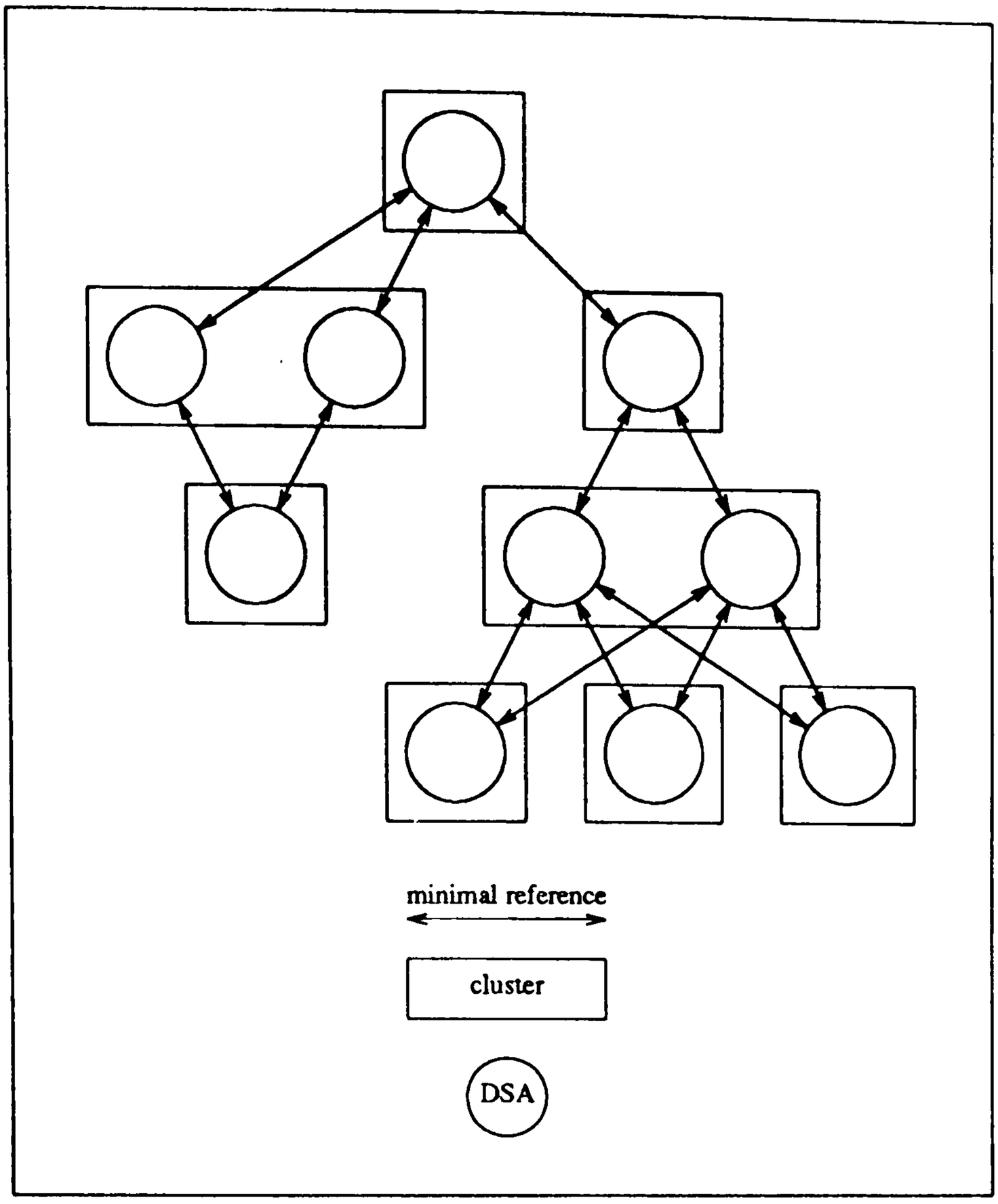


Figure 6.4: DSA clusters and the revised structure of minimal knowledge

The restructuring of minimal knowledge increases the robustness of navigation and hence the operation of the Directory as a whole. In general, the navigation step procedure operates as before except that it returns the references of several best match closer DSAs belonging to the same cluster. If a DSA attempts to navigate to an immediately superior or inferior DSA and finds that it is unobtainable, it may retry at another DSA in the same cluster.

Opportune knowledge may be extended in a similar way to take account of clustering. Whenever a DSA's opportune knowledge references a remote DSA it could also reference the other DSAs within its cluster. This suggests that operation and result envelopes should be extended to include the references of all DSAs in a cluster in order to propagate knowledge of entire clusters as opposed to single DSAs.

The knowledge management mechanisms of section 5.3 also require adaptation to the revised structure of minimal and opportune knowledge. In particular, the effects of the

Update Reference and *Delete Reference* operations should be propagated to all DSAs within the cluster of their target DSA.

6.4.2. The execution of updating operations

The introduction of DSA clusters has an impact on the execution phases of those operations updating directory information. In general, an update operation is first navigated to one of the DSAs from the cluster responsible for the target entry. This DSA is the coordinating DSA.

The coordinating DSA is responsible for executing the operation as described in chapter 5. This includes checking access and integrity controls and interacting with remote DSAs when necessary. In addition, the coordinating DSA assumes responsibility for updating the other DSAs within its cluster via the local multiple master update policy.

Summary

In conclusion, the clustering of DSAs increases the robustness of the Directory service by introducing multiple mastership of DIT fragments, but on a limited scale, thus minimising consistency overheads. Clusters will typically be defined on an organisational basis and will contain a small number of DSAs within a reliable communications environment. The information shared by DSAs within a cluster must be maintained by a multiple master update policy supporting strong consistency. The exact choice of policy may vary from cluster to cluster.

One major advantage of clustering is the resulting increase in robustness of the navigation algorithm which is fundamental to the entire distributed operation of the Directory. This requires extensions to the structures of minimal and opportune knowledge as specified by chapter 5. These extensions also require changes to the directory knowledge management mechanisms.

6.5. Replication between DSA clusters

Section 6.1.2 described example uses of replication within the Directory service. The common feature of these is that a DSA uses replication to increase its knowledge of the Directory beyond those fragments for which it is responsible as a master. The extra information acquired can be used to increase the efficiency of the Directory by reducing the number of DSA interactions required to execute an operation. Replication within DSA clusters (as described above) does not increase the information known by any one DSA and therefore

cannot satisfy this motivation. Instead, replication to increase a DSA's information base must occur between DSAs from different clusters. This inter-cluster replication is supported by a *single master* update mechanism where the single master is, in fact, a DSA cluster.

Replication between clusters occurs when a single slave DSA from one cluster establishes a *replication agreement* with all DSAs from another cluster (called the *master cluster*). The replication agreement specifies the information to be replicated by the slave DSA and sets various controls governing the replication process. More precisely, it specifies the initial information to be replicated, a policy for determining which future information should be replicated and whether updates will be *master initiated* or *slave initiated*. It should be noted that the slave DSA acts independently of other DSAs within its cluster.

This section addresses several issues concerning inter-cluster replication supported by single master update mechanisms.

- Section 6.5.1 examines the granularity of replication.
- Section 6.5.2 discusses *replication policies* describing replicated information.
- Section 6.5.3 specifies the structure of replicated knowledge.
- Section 6.5.4 discusses the management of replication agreements.

6.5.1. Granularity of replication

Specifying a suitable granularity for replicated information is an important aspect of the directory replication model. Replication agreements could allow various granularities:

- DSAs could replicate individual attributes from directory entries. However, most operations access single or multiple entries and an attribute level granularity therefore seems unnecessarily complex, resulting in a large management overhead.
- The entry could form the basic unit of replication. However, it is clearly useful to replicate sets of entries under a single agreement.
- The master fragment (see section 5.2) could form the unit of replication. However these fragments will probably be very large and therefore too unwieldy.

The approach proposed by this thesis defines the entry as the smallest possible unit of replication and the fragment as the largest. In general, a slave DSA copies "useful" subsets of entire fragments from a cluster of master DSAs. Useful subsets can be defined in terms of *Object Set Descriptors* (OSDs) as specified in section 4.2.6. To recap, an Object Set Descriptor describes a set of entries, called its *object set*, in terms of the root of a DIT subtree, an indication of depth of the subtree and a *filter* describing attributes possessed by entries belonging to the object set. Thus, OSDs provide a general and flexible mechanism

for describing sets of entries and therefore replicated information. For example, we could replicate all distribution list entries in a master fragment using an Object Set Descriptor naming the root of the fragment and giving a filter matching *object class = distribution list*.

At this point, it is worth clarifying some terminology. The remainder of this chapter uses the following definitions.

- A *master fragment* is a fragment of the DIT which is the responsibility of a master DSA. Master fragments were defined in chapter 5 and may be referred to simply as *fragments*.
- A *replicated fragment* is a set of entries replicated between DSAs from different clusters. Replicated fragments are derived by enumerating OSDs within master fragments. They are, therefore, subsets of master fragments.

It is important to note that an OSD defining a replicated fragment does not cross master fragment boundaries (i.e. is only enumerated within one master fragment and therefore one DSA).

There is a further, orthogonal, constraint which can be placed on replicated fragments. Replication might be intended to speed up navigation and name verification. This only requires the replication of the relevant DIT naming structure and not the replication of general attributes. It should therefore be possible to limit replicated information to naming information only, as opposed to entire entries, thus avoiding the costly maintenance of replicated attributes. This constraint defines *replicated* knowledge and is discussed in section 6.5.3 below.

6.5.2. Statement of replication policy

The use of Object Set Descriptors to specify replicated fragments gains us more than just a convenient set description technique. Object Set Descriptors are a logical description of information and therefore provide a statement of *replication policy* (i.e. the intented types of replicated information). Thus, the administrator initiating a replication agreement supplies a general description of the information to be copied. A master DSA is then able to determine the entries satisfying this description at any given time. This is important because it enables the system to automatically respond to changes in the DIT and decide which new information should be replicated without human intervention.

For example, if the replication policy was stated by the example OSD in the previous section and a new *distribution list* were added to the relevant subtree of the DIT, the master DSA would know to replicate it. On the other hand, if a new *person* entry were added, the master DSA would know not to replicate it.

The ability to specify general policies within replication agreements is crucial because it allows the Directory system to automatically manage replicated information as the DIT changes.

6.5.3. Replicated knowledge

This section discusses the structure of replicated knowledge as specified by section 5.2 of this thesis. Replicated knowledge is the third category of knowledge available to a DSA during navigation and, like opportune knowledge, is used to improve the robustness and efficiency of the navigation process.

Replicated knowledge is included within replicated fragments.

Object Set Descriptors are used to specify replicated fragments. When replication occurs these OSDs are mapped onto a number of *replicated subtrees* which are copied from master to slave DSAs. These subtrees are *incomplete* subtrees because the DIT may extend upwards and downwards beyond them. In order to use the replicated knowledge inherent in a replicated fragment, a DSA must know whether each apparent leaf entry in a replicated subtree is a true leaf in the DIT or whether the DIT continues beyond the subtree. In general, for each replicated subtree the slave DSA holds a *replication knowledge tree* containing replicated entries and a number of *continuation points* indicating where the DIT continues. The structure of a replication Knowledge tree is shown in figure 6.5

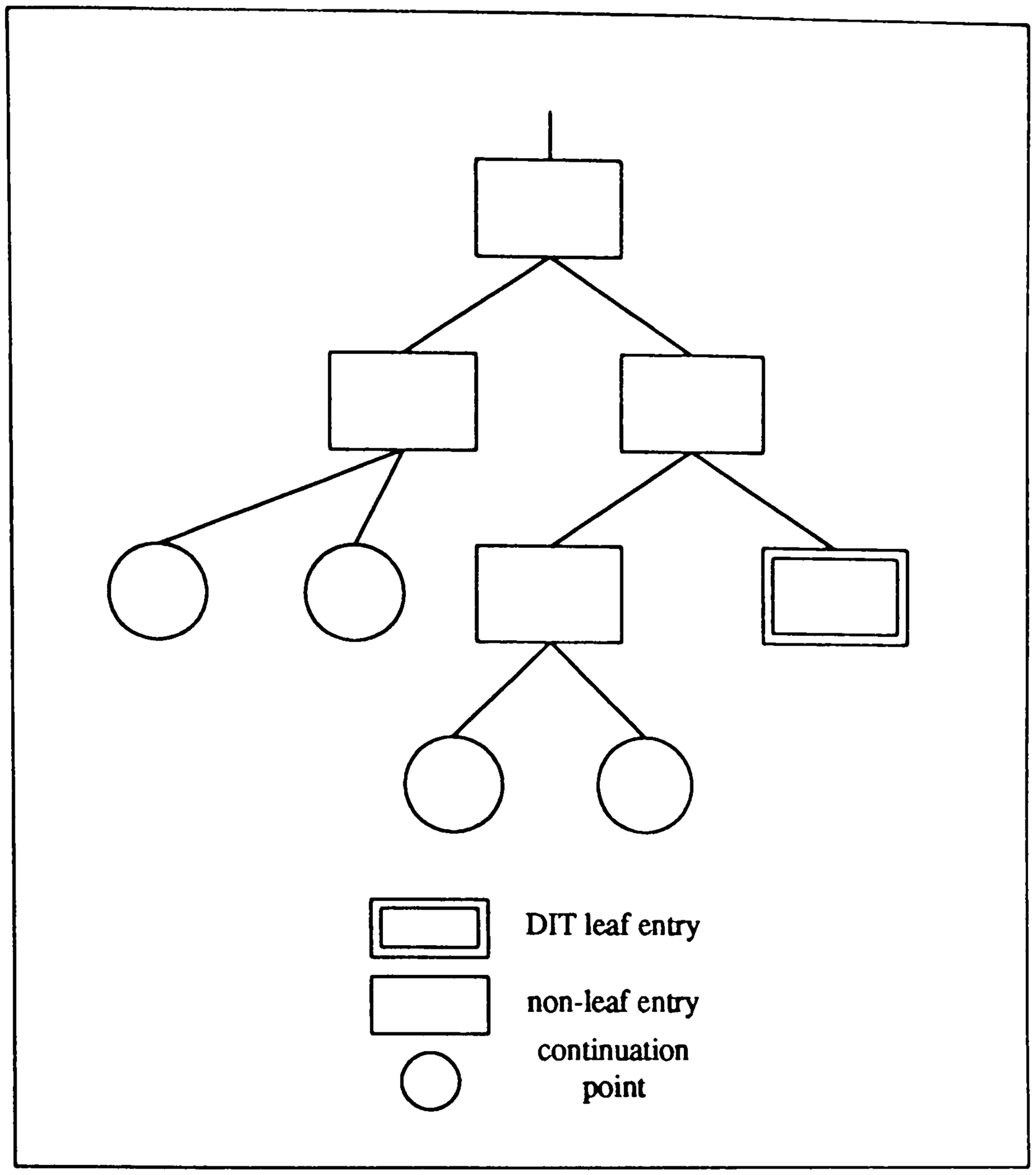


Figure 6.5: Example replication knowledge tree

Continuation points are associated with the names of entries immediately subordinate to the replicated subtree and may optionally include references to the DSAs responsible for these entries. It is these references which define the replicated knowledge inherent in the replicated fragments. The structure of a replication knowledge tree is specified by the following ASN.1 structure.

```
ReplicationKnowledgeTree ::= SET {
                entries [0] SET OF EntryInformation,
                continuations [1] SET OF ContinuationPoint,
                type [2] INTEGER { full (0), naming (1)} }

ContinuationPoint ::= SEQUENCE {
                name DistinguishedName,
                Reference OPTIONAL }
```

This structure also indicates whether full entries are replicated or whether only naming attributes are present. In the latter case, the replication knowledge tree contains only replicated knowledge.

In order to perform name verification, a DSA must understand how replicated sets of entries (and therefore replicated knowledge) relate to the overall structure of the DIT. In particular, it must be able to determine whether a purported name falls inside or outside the scope of replicated fragments. This can be achieved by inspecting replication knowledge trees using the following steps.

- If the name associated with a continuation point is a prefix of the purported name, then the purported name is outside the scope of the replicated fragment.
- If there is no continuation point that is a prefix of the purported name and the root of the replication knowledge tree is a prefix, then the purported name is within the scope of the replicated fragment. The replicated fragment can then be used during name verification and navigation.

The navigation and name verification procedures of chapters 3 and 5 require extension to support the use of replicated information using these steps.

In summary, replicated knowledge is inherent in the structure of replicated fragments. This knowledge may be used for navigation and name verification provided that replicated fragments include *continuation points* indicating where the DIT extends beyond them.

The management of replicated knowledge occurs during the general management of replicated fragments. This is discussed in the following section.

6.5.4. Managing replication agreements

This section discusses the management of replication agreements between DSAs and the maintenance of replicated information. Although this chapter does not specify a detailed replication mechanism, it is concerned with the general support required for replication. This work therefore considers the support required for the following issues.

- Establishing single master replication agreements between DSAs.
- Maintaining replicated information under replication agreements.
- Terminating replication agreements.

The required support is expressed in terms of several new abstract operations forming the basis of a "directory replication management protocol" (see Appendix A). The following table lists the operations defined by this section. It should be noted that this work only specifies the basic support required for replication and may require future extension.

Replication management operations	
operation	meaning
Establish Replication	Establish a new replication agreement
Terminate Replication	Terminate an existing replication agreement
Update Replication	Master initiated update of replicated information
Refresh Replication	Slave initiated update of replicated information

Establishing replication agreements

Replication agreements are initiated by a slave DSA making a request to one DSA within a master cluster. The slave DSA specifies the proposed replication agreement in terms of a set of Object Set Descriptors indicating the replication policy, whether naming information or full entries are to be replicated and whether updates are to be master or slave initiated. Master initiated update represents a normal single master replication policy whereas slave initiated update represents a snapshot policy.

On receipt of the proposed agreement, the master DSA decides whether it accepts or rejects the proposal (this decision is a local matter). If rejected, a suitable error is returned. If accepted, the following steps are performed:

- It enumerates the supplied OSDs within its master fragments, thus generating a set of replication knowledge trees.
- It returns these replication knowledge trees to the slave DSA. It also returns the references of ALL DSAs in the master cluster as well as a unique identifier referring to this agreement.
- The master then stores details of the new agreement as well as local copies of the replication knowledge trees.
- The master informs all other DSAs in the master cluster of the new agreement and its parameters, including the reference of the slave DSA. This requires a multiple master update within the cluster.
- The other master DSAs calculate and store the replication knowledge trees for this agreement.

Replication agreements are established by the *Establish Replication* operation as specified by the following ASN.1 structure.

```
EstablishReplication ::= ABSTRACT-OPERATION
        ARGUMENT EstablishReplicationArgument
        RESULT EstablishReplicationResult
        ERRORS { DSAError, EstablishError }
```

```
EstablishReplicationArgument ::= ReplicationAgreement

EstablishReplicationResult ::= SET {
        identifier [0] INTEGER,
        information [1] SET OF ReplicationKnowledgeTree,
        master-dsas [2] SET OF Reference }
```

This operation is called between DSAs and therefore belongs to the DSP. However, it has a counterpart belonging to the DAP allowing administrators to initiate the replication process. The DAP version specifies the names of those DSAs involved in the agreement and returns an indication of success or failure as opposed to the replicated information.

A replication agreement is represented by the following ASN.1 structure.

```
ReplicationAgreement ::= SET {
                policy [0] SET OF ObjectSetDescriptor,
                information [1] INTEGER { full (0), naming (1) },
                update-style [2] INTEGER { master (0), slave (1) } }
```

Following the establishment of an agreement, the DSAs belonging to the master cluster should store the information listed below.

- The agreement ID.
- The reference of the slave DSA.
- The OSDs specifying the replication policy.
- Replication knowledge trees derived from the policy specification.
- Indications of whether updates will be slave or master initiated and whether full entries or just naming structures are to be replicated.

The slave DSA stores the following information:

- The agreement ID.
- The references of all DSAs in the master cluster.
- The OSDs specifying the replication policy.
- The replicated information in the form of a set of replication knowledge trees.
- Indications of whether updates will be slave or master initiated and whether full entries or just naming structures are to be replicated.

It should be noted that both the master and slave DSAs must maintain copies of the current replication knowledge trees for the agreement in order to determine when updates should be applied and for crash recovery purposes.

The process of establishing and maintaining a replication agreement requires that master DSAs are able to map a set of OSDs into a set of replication knowledge trees. This requires

enumeration of OSDs (i.e. determining the names of all entries satisfying them) which, in turn, requires enumerating all entries in a subtree of the DIT and checking whether they satisfy a filter. It should be noted that enumeration only occurs at a single DSA because replication agreements cannot cross master fragment boundaries. Therefore, it is not likely to be prohibitively expensive or time consuming as it requires no DSA interactions.

Master initiated update

Once a replication agreement has been established, replicated information will require updates following changes to master fragments. Updates may be master or slave initiated.

Master initiated update of replicated information requires that a master DSA performing a directory update propagates it to all relevant slave copies of information. Thus, on receipt of an update, the master first updates all other masters in the cluster and then examines local copies of replication agreements. For those agreements specified as having master initiated update, the master determines whether the slave requires updating and, if so, propagates the updates.

The master DSA decides whether a given slave requires updating by comparing its updated master fragment with its copies of the replication knowledge trees for each agreement. Updating should occur under the following circumstances:

- If the update modifies or deletes information belonging to replication knowledge trees held by the slave.
- If the update adds a new entry or alias satisfying the OSDs specifying the replication policy. This is the condition allowing the Directory to automatically decide which new information should be replicated to which slaves.

A master DSA updates a slave DSA by the *Update Replication* operation specified below.

```
UpdateReplication ::= ABSTRACT-OPERATION
        ARGUMENT UpdateReplicationArgument
        RESULT UpdateReplicationResult
        ERRORS { DSAError, ReplicationError }

UpdateReplicationArgument ::= SET {
        agreement-id [0] INTEGER,
        updates [1] SET OF Update }

Update ::= CHOICE {
        add-entry [0] EntryInformation
        add-alias [1] AliasInformation,
        delete-entry [2] DistinguishedName
        add-continuation [3] ContinuationPoint,
        delete-continuation [4] DistinguishedName,
        modify-attributes [5] Modification }
```

```
UpdateReplicationResult ::= NULL
```

This operation belongs to the DSP and allows the master DSA to add, delete and modify entries and aliases covered by the identified agreement. The master DSA also updates continuation points.

Slave initiated update

Slave initiated update requires that a slave DSA periodically refreshes replicated information. The slave can quote the agreement ID to any DSA from the master cluster which may then recalculate the replication knowledge trees from the OSDs specifying the replication policy and return updates to the slave as necessary. Slave initiated update is achieved by the *Refresh Replication* operation specified below.

```
RefreshReplication ::= ABSTRACT-OPERATION
        ARGUMENT RefreshReplicationArgument
        RESULT RefreshReplicationResult
        ERRORS { DSAError, ReplicationError, UpdateError }

RefreshReplicationArgument ::= SET {
        agreement-id [0] INTEGER }

RefreshReplicationResult ::= SET OF Update
```

Terminating replication

Replication agreements may be terminated by either the slave DSA or any DSA from the master cluster. Termination requires that all DSAs involved in an agreement delete all information specifically concerned with that agreement and is achieved by the *Terminate Replication* operation.

```
TerminateReplication ::= ABSTRACT-OPERATION
        ARGUMENT TerminateReplicationArgument
        RESULT TerminateReplicationResult
        ERRORS { DSAError, ReplicationError, UpdateError }

TerminateReplicationArgument ::= SET {
        agreement-id [0] INTEGER }

TerminateReplicationResult ::= NULL
```

Like the *Establish Replication* operation, this operation has a DAP counterpart allowing administrators to request the termination of agreements.

6.5.5. Summary of replication between clusters

Replication between clusters is based on a *master/slave* update mechanism where the master may be any DSA from a master cluster and the slave is a DSA from another cluster.

Replication is described by replication agreements specifying the information to be copied and whether updates should be master or slave initiated. The management of replication agreements is supported by a number of operations allowing them to be established and terminated and also implementing master or slave driven updates.

Replicated information is specified in terms of *Object Set Descriptors* representing a logical statement of replication policy allowing the Directory system to automatically manage copies as the DIT changes.

6.6. Crash recovery

Both the single and multiple master replication mechanisms described by this chapter cannot be guaranteed to leave directory information in a consistent state in the event of a DSA or communications link crashing at a vital moment. In general, there is no protocol able to guarantee that all the database nodes involved in a transaction will either "roll back" or commit in unison in the face of arbitrary failures [DATE83]. Imagine there was such a protocol, and that it required N (N > 0) essential intersite messages for completion. Suppose that the last message (N) was lost due to node or communication failure. Then either message N wasn't essential (contradiction), or the protocol fails (contradiction). Depressing as this result may be, the Directory system must be prepared for failures and should be able to recover from inconsistencies introduced by DSA or communication link crashes. This section therefore discusses the need for crash recovery mechanisms within the Directory service.

The failure of DSAs or communication links could have many effects on the operation of the Directory service. The following examples should be sufficient to indicate the general problems involved.

1. Parts of the DIT may become separated for a period of time. In particular, the Directory may become separated into a number of smaller, operational sub-directories unable to interact.
2. Operations will be unable to access entries belonging to unobtainable DSAs.
3. The propagation of entry and attribute definitions may be interrupted.

4. The update of replicated information may become impossible.

Previous work in the field of distributed databases has shown that resilient crash recovery is difficult to implement and that the independent recovery of a single node following the failure of two or more other nodes is impossible to guarantee. However, there are a number of heuristic approaches to preventing and recovering from the above problems.

The first case can be largely avoided by the sensible use of opportune and replicated knowledge to increase the robustness of the Directory system. These categories of knowledge reduce the dependency on any one DSA for navigation and therefore reduce the chances of the Directory system becoming separated.

The replication of information between DSA clusters increases its availability for retrieval and thus partially eases the second problem. The replication of information within DSA clusters also reduces the effects of the second problem by increasing the availability of information for update.

The third and fourth problems require support for crash recovery techniques within the Directory system. Let us consider the case where a DSA crashes and is unobtainable for a period of time. This DSA may belong to a master cluster for some information and, in addition, may act as a slave for other information. During its period of inactivity, several updates may be committed by the other DSAs within its cluster and at the clusters with which it has replication agreements, leaving the DSA with inconsistent information. Crash recovery requires that, upon successful restart, the DSA must resynchronise with all other relevant DSAs and restore its information to a consistent state before answering any general directory queries. In addition, DSAs remaining operational should timeout and retry updates until recovery is complete. Resynchronisation could occur in two stages:

- Firstly, the DSA contacts an available DSA within its own cluster and requests all missed updates for master fragments. Provision of missed updates requires either that transaction logs are maintained by master DSAs or that the DSA re-reads and compares all fragments for which it is responsible.
- Secondly, the DSA must attempt to restore all slave copies of information by requesting a *Refresh Replication* operation at an available DSA from the master cluster. This is equivalent to a normal slave initiated update. The master DSA can calculate the necessary updates by comparing replication knowledge trees and master fragments in the usual way.

These strategies will often cope with recovery from a DSA crash. However, the failure of a communication link, where the disconnected DSAs remain operational, presents a more difficult problem. For example, two DSAs from a master cluster may become disconnected

but may continue to accept updates independently. Resynchronisation of such DSAs is difficult and may involve complex negotiations or even human intervention.

Approaches to crash recovery can be adapted from general distributed systems. These include the following:

- The coordinating DSA for a multiple master update should perform appropriate timeout checks and retransmit messages as many times as necessary to ensure that they eventually reach their intended destination.
- Following DSA failure, a DSA should communicate with other DSAs to restore consistency of information.
- DSAs should maintain transaction logs specifying the history of changes to directory information.

Similar mechanisms are reviewed in [KOH81] and could be incorporated within the Directory service.

6.7. The impact of replication on distributed operations

This section summarises the impact of replication on the distributed model of chapter 5. In particular, it considers the effect of replication on knowledge, navigation and the distributed execution of operations. Much of this work is included in other sections of this chapter. However, for reasons of clarity, it is summarised below in a concise form.

Impact on partitioning, knowledge and navigation

Section 5.2 specified the partitioning of the Directory Information Tree between a set of autonomous DSAs and developed a knowledge model describing the configuration of these DSAs at any one time. The introduction of DSA clusters means that the DIT is effectively partitioned between clusters as opposed to single DSAs. This requires a number of adjustments to the directory knowledge model.

- The structure of minimal knowledge is extended to include references to all DSAs in immediately superior and inferior clusters (see figure 6.4).
- Opportune knowledge may also be extended to include references to all DSAs within a cluster, although this is not strictly necessary.
- The structure of replicated knowledge is specified in terms of the *replication knowledge trees* acquired during a replication agreement between DSAs from different clusters.

This enhanced knowledge model has a subsequent impact on the navigation of operations. In general, the robustness of navigation is improved by the clustering of DSAs. Each navigation step may return the references of several best match DSAs, including more than one minimal knowledge reference. If a DSA is unobtainable, another may be tried until contact is established.

In addition to an improvement in navigation, local name verification may also utilise the naming structure inherent in replication knowledge trees to improve the overall performance of distributed name resolution.

Knowledge management and distributed operations

The management of knowledge is vital to the successful distributed operation of the Directory. The introduction of replication and DSA clusters has the following impact on the knowledge management techniques of section 5.3.

- The DSAs within a cluster share minimal knowledge and are responsible for updating references via the local multiple master concurrency control mechanism. Thus, on receipt of an *Update Reference* or *Delete Reference* operation, a DSA must propagate the change in minimal knowledge to all the DSAs within the local cluster. The remote DSA initiating the operation only needs to update a single DSA from this cluster.
- Opportune knowledge is not shared within a cluster. Consequently, the navigation error detection and correction mechanisms of section 5.3.3 are unaffected by the directory replication model.
- Replicated knowledge is managed as part of the general update of replicated information. This is supported by the single master concurrency control mechanism.

Replicated information has a major impact on the execution of operations as described by section 5.4. Firstly, the navigation phase of each operation may be shortened by the increase in efficiency of the navigation procedure as mentioned above. Secondly, the execution phases of operations may be altered. It is possible to distinguish two general cases.

The execution phases of those operations retrieving information may be simplified. For example, the coordinating DSA for a List Subordinates operation may replicate the entries of its immediate children, thus guaranteeing local execution of the operation. Alternatively, the number of DSAs involved in a recursive search may be drastically reduced by the replication of lower subtrees of the DIT within a given DSA.

Conversely, the execution phases of those operations updating information are complicated by the introduction of replication. Typically, these execution phases progress as before with the addition of a multiple master update to all DSAs within the cluster of the coordinating

DSA. The manipulation of entry and attribute definitions may require many multiple master updates as changes are propagated to the DSA clusters responsible for a DIT subtree.

6.8. Summary of the directory replication model

This chapter has proposed a general model supporting replication within the Directory service. This model provides an abstract framework describing replication issues and might support many different implementations of directory replication mechanisms.

The proposed model is based on the concept of *DSA clusters* reflecting the different motivations for replication within the Directory service. A DSA cluster is a tightly bound set of DSAs such that:

- The DSAs within a cluster are masters for the same DIT fragments and are tightly bound by a multiple master concurrency control mechanism. Replication within a cluster increases the availability of information for update and therefore improves the robustness of the Directory.
- Replication may occur between DSAs from different clusters under a single master or snapshot concurrency control mechanism. Thus, clusters may be loosely bound to each other increasing the availability of information for retrieval and, consequently, the efficiency of the Directory system.

The choice of multiple master concurrency control mechanism is local to each DSA cluster. However, this chapter reviewed several approaches within distributed databases, concluding that *timestamp* based mechanisms seem to offer the best general solution.

This chapter examined several aspects of single master replication between clusters.

- The structure of replicated knowledge was derived from the general structure of replicated DIT fragments. Furthermore, it was concluded that replicated knowledge is managed as part of the general replication process.
- The use of *Object Set Descriptors* to describe replicated information was discussed. The discussion concluded that OSDs provide a useful tool for describing information in generic terms and therefore representing *replication policies*.
- Support for the management of *replication agreements* between DSAs was considered and several new abstract operations were proposed, forming the basis of a replication management protocol. These operations also support the update of replicated information following the update of master DIT fragments.

This chapter also considered the impact of replication on the distributed model of chapter 5. Finally, the requirement for crash recovery techniques within the Directory service was noted.

This concludes the discussion of the local conceptual layer and also of the specification and modelling work within this thesis. Chapter 7 describes the implementation work occurring within this research. Much of this work concerns the implementation of a Directory System Agent and therefore belongs to the local internal layer.

Chapter 7

The Local Internal Layer: A Directory Implementation

The implementation of a prototype Directory service has formed an important part of this research. Although this thesis has emphasised the development of several abstract Directory models, implementation work has consumed a considerable amount of time and effort. Furthermore, the prototype has provided a test-bed for many of the models and concepts described by the previous chapters and has therefore been an integral part of the design process.

This chapter describes the implementation of a prototype DSA and several DUAs. In particular, it considers the use of existing software tools in their construction. A major part of this work concerns the mapping from the directory local conceptual model to a specific local internal model based on the relational data model.

The work within this chapter provides an overview of the pilot implementation without describing its coding in detail. It also presents the insights and conclusions drawn from the prototyping experience.

- Section 7.1 describes the background to implementation work in terms of motivations, constraints, functionality and, finally, a review of supporting software tools.
- Section 7.2 presents the functional architectures of the pilot DSA and DUAs and explains how the supporting tools are integrated into the prototype system.
- Section 7.3 describes the implementation of a Directory System Agent. It examines the different modules involved in its operation and presents the mapping from the local conceptual model to the relational data model.
- Section 7.4 outlines the implementation of a basic DUA support library and demonstrates its use in constructing a range of DUAs. A specific DUA for handling distribution lists is considered as an example.
- Section 7.5 summarises implementation work and presents its results and conclusions.

This thesis has adopted a top down approach to describing the Directory service and its structure does not always reflect the chronological progression of my work. In reality, specification and implementation work occurred in parallel resulting in minor differences

between the final specification and implementation. These differences are noted where appropriate.

7.1. Background to implementation work

This section outlines the motivations and constraints defining the implementation work described by this chapter. It also discusses the broad goals and functionality of the prototype system and provides an overview of the supporting *RTI Ingres* relational database [RTI85] and *ISO Development Environment* [ROSE88] software tools.

7.1.1. Motivations

There have been several motivations for undertaking implementation work within this research.

- Prototyping has formed an integral part of the design process. It has tested many of the concepts developed by specification work and has indicated possible shortcomings and flaws. The results of prototyping have then been re-input to the specification process in order to refine the various directory models.
- Prototyping has shown how existing software tools might be used to support Directory services. Similar tools may be available within many environments in the near future.
- Prototyping has indicated issues to be addressed by production Directory services.

Provision of a working Directory for non-experimental use, even on a limited scale (e.g. department wide), has not been a motivation for this work.

7.1.2. Pragmatic constraints and their implications

Prototyping has been subject to various pragmatic constraints affecting the goals and functionality of the pilot system.

A major constraint was the limited time and manpower available for this research, rendering a full implementation of the specified Directory service impossible. In particular, realisation of the full complement of distributed directory operations was not feasible. It is worth bearing in mind that the implementation of full scale Directory services is currently being undertaken within large, international projects using teams of researchers.

A second constraint was the lack of network resources with which to implement a large scale distributed system. This manifested itself in several ways.

- Prototyping was allocated a single fileserver and two workstations on a local area network.
- Only a small number of departmental machines supported an OSI based network environment, capable of running a Remote Operations Service and, hence, the prototype Directory service.

The limited availability of resources ensured that a fully distributed prototype was not feasible. Implementation work has therefore been concerned with developing a stand alone DSA supporting information management tools. Later work extended this to include limited facilities for interaction with remote DSAs.

7.1.3. Functionality of the prototype Directory

The previous sections described the motivations and pragmatic constraints affecting the design of the prototype Directory service. This section outlines the broad functionality of the prototype.

DSA functionality

The main aspect of implementation work has been the design of a Directory System Agent interacting with several different DUAs. This DSA supports the directory information model in terms of the Directory Information Tree, entries, attributes and aliases. It provides a Directory Access Protocol allowing the retrieval of information, browsing the DIT and the general modification of information. The following operations are supported by the prototype.

Implemented operations
Read Entry
List Subordinates
Modify Entry
Add Entry
Delete Entry
Add Alias
Delete Alias
Add Attribute Def
Delete Attribute Def
Read Attribute Def
Add Entry Def
Delete Entry Def
Read Entry Def

The *Search* and *Suspend* operations are not supported.

The prototype DSA supports the information management tools specified by chapter 4. This includes an access control mechanism based on Access Control Lists and an integrity mechanism based on entry and attribute definitions. The integrity mechanism allows the dynamic management of entry and attribute definitions, although different attribute syntaxes and value constraints are not yet implemented.

The prototype DSA uses the RTI Ingres relational database management system as an information storage and retrieval mechanism and a major part of implementation work has involved the mapping from the directory information model to the relational model [CODD70]. This has included mapping directory operations to relational queries.

In addition to its stand alone mode, the prototype DSA supports limited distributed operation. This takes the form of a distributed *Read Entry* operation allowing DSAs to interact in chaining or referral modes when retrieving information. The Read Entry operation uses an implementation of the navigation and name verification algorithms. There was not sufficient time to implement distributed versions of the remaining directory operations. However, the Read Entry operation is sufficient to illustrate the general principles of navigation and distributed execution. The implementation of the prototype DSA is described in section 7.3.

DUA functionality

Three DUAs have been implemented during this research. The first is a simple interactive DUA supporting the testing of the prototype system. The others have been implemented in cooperation with the AMIGO MHS+ project in order to test a distribution list protocol using both an X.400 Message Handling Service and the prototype Directory service. This work is outlined in section 7.4 and has been fully specified in [BENF87] and [BENF88a].

The Directory Access and System Protocols are supported by the ISO Development Environment (ISODE). This enables the Directory to operate using OSI protocols and, in particular, the *Remote Operations Service*. The next section provides an overview of ISODE and the RTI Ingres database system.

7.1.4. Tools supporting prototyping

The bulk of the prototype Directory service is written in the C programming language [KERN78] and runs under SUN 3.4 UNIX† on a SUN 3/160 fileserver and 3/50 workstations. The prototype makes use of two additional software tools:

---

† UNIX is a trademark of Bell Laboratories.

- The *RTI Ingres* relational database management system provides back-end storage within the DSA.
- The *ISO Development Environment* (ISODE) provides an OSI protocol stack, Remote Operations Service and several other useful features. ISODE is used by both DUAs and DSAs for communication support.

The following sections briefly describe these tools and explain the reasons for their use.

The RTI Ingres RDBMS

RTI Ingres is a commercial relational database management system (RDBMS) [ROWE86], developed from *University Ingres*, originally written by Stonebraker and Wong [STON85, EPS77].

The prime motivation for using a relational database for prototyping is the freedom that it provides. The relational model is both flexible and easy to use, facilitating the fast development and alteration of prototypes. In addition, the relational model provides a further layer of indirection between the directory model and physical storage and access methods. This shields physical data from the many changes made to the prototype during its implementation.

A second motivation for the use of an RDBMS is the provision of transaction control, logging and crash recovery mechanisms which may be used to solve similar problems within the prototype Directory.

Finally, the aim of prototyping was to explore issues relevant to Directory services and not to re-invent the database. It may be that, for efficiency reasons, production directories will use their own specifically tailored storage mechanisms. However, implementation of such mechanisms is not consistent with the goals and constraints of this work.

The RTI Ingres database is currently one of the better relational systems available and was chosen for the following reasons:

- It supports both the SQL [IBM81] and QUEL [STON85] relational query languages allowing extremely complex queries to the database.
- It supports the embedded SQL/C and QUEL/C programming interfaces allowing database queries to be incorporated into C programs.
- It allows character fields of up to 2000 characters long. Many systems impose much stricter limitations.
- It supports user specified transaction control in terms of savepoints and COMMIT and ABORT statements.

- It offers a wide choice of physical access methods and indexing techniques.

The use of the RTI Ingres database will be described in section 7.4. The remainder of this chapter assumes that the reader is generally familiar with the relational data model.

The ISO Development Environment (ISODE)

ISODE has been developed by Marshall Rose, in association with a number of other researchers, [ROSE88, ONIO88] with the intention of providing a general OSI programming environment. This environment includes a number of tools facilitating the development of applications supported by standard OSI protocols. ISODE tools take the form of a set of libraries, included within C programs, allowing programmers to use ISO transport, session, presentation and application layer protocols and services.

These protocols may run over the ISO X.25, internet TCP/IP [DDN-TCP] and internet UDP/IP [DDN-UDP] network protocols as well as over the ISO TP4 [ISO-TP4] transport protocol. This allows OSI style applications to bridge the gap between a number of network communities.

The prototype Directory service utilises the following ISODE tools to establish connections between DUAs and DSAs, to invoke remote operations and to encode/decode ASN.1 data structures to/from C data structures within application programs.

- The *tsap* library implementing the transport layer protocol.
- The *ssap* library implementing the session layer protocol.
- The *psap* library implementing the presentation layer protocol.
- The *acsap* library responsible for *association control*. This establishes and manages a client-server connection between two applications (e.g. between DUAs and DSAs).
- The *rosap* library implementing the *Remote Operations Service* (ROS) allowing a client to invoke operations at a remote server.
- The *Pepy* generator/parser mapping between ASN.1 data structures and C data structures.

The Pepy program is of particular importance to the prototype Directory. Pepy is a YACC based parser able to parse and build ASN.1 data structures from C data structures. It allows the programmer to input an ASN.1 definition file, annotated with C statements, and uses this file to automatically generate parsing routines. The use of Pepy and other tools will be explained as they are encountered.

This concludes the review of those software tools used by the prototype Directory service. The following section describes the functional architectures of the DSA and DUA components of the directory implementation. These form the basis for sections 7.3 and 7.4 which discuss the DSA and DUA implementations in greater detail.

7.2. Functional architectures of the DSA and DUA

In order to systematically describe the implementation of the prototype Directory service, DSA and DUA functional architectures are required to provide a reference model for later sections. These functional architectures are presented below. They outline the broad structure of the DSA and DUA entities and illustrate the use of the RTI Ingres and ISODE tools in their implementation.

7.2.1. Functional architecture of the DSA

The structure of the DSA may be divided into two functional elements as shown in figure 7.1.

- The *dispatcher* [1] manages communication with DUAs and other DSAs.
- The *executor* [1] executes operations on local information.

The dispatcher

The dispatcher manages associations with DUAs. It receives association and disconnection requests and maintains association information in a local table. During the course of an association, a DUA requests a number of operations at the DSA. Operation arguments and results are transferred as ASN.1 data structures. However, operations are executed by C routines which view arguments and results in terms of C data structures. The dispatcher is therefore responsible for decoding ASN.1 arguments into C data structures to be passed to the executor and encoding the results from C structures back into ASN.1.

The dispatcher also manages associations with other DSAs during the course of distributed Read Entry operations, when it is responsible for generating association requests and for requesting remote operations.

The dispatcher may be subdivided into two elements:

---

[1] These terms were defined in the Melbourne X.ds output [CCITT-XDS86].

- The *ROS manager* manages associations, receives operations and dispatches results. It utilises the ISODE *acsap*, *rosap*, *psap*, *tsap* and *ssap* libraries.
- The *encoder-decoder* maps between ASN.1 data structures and C data structures. It utilises the ISODE *Pepy* generator/parser.

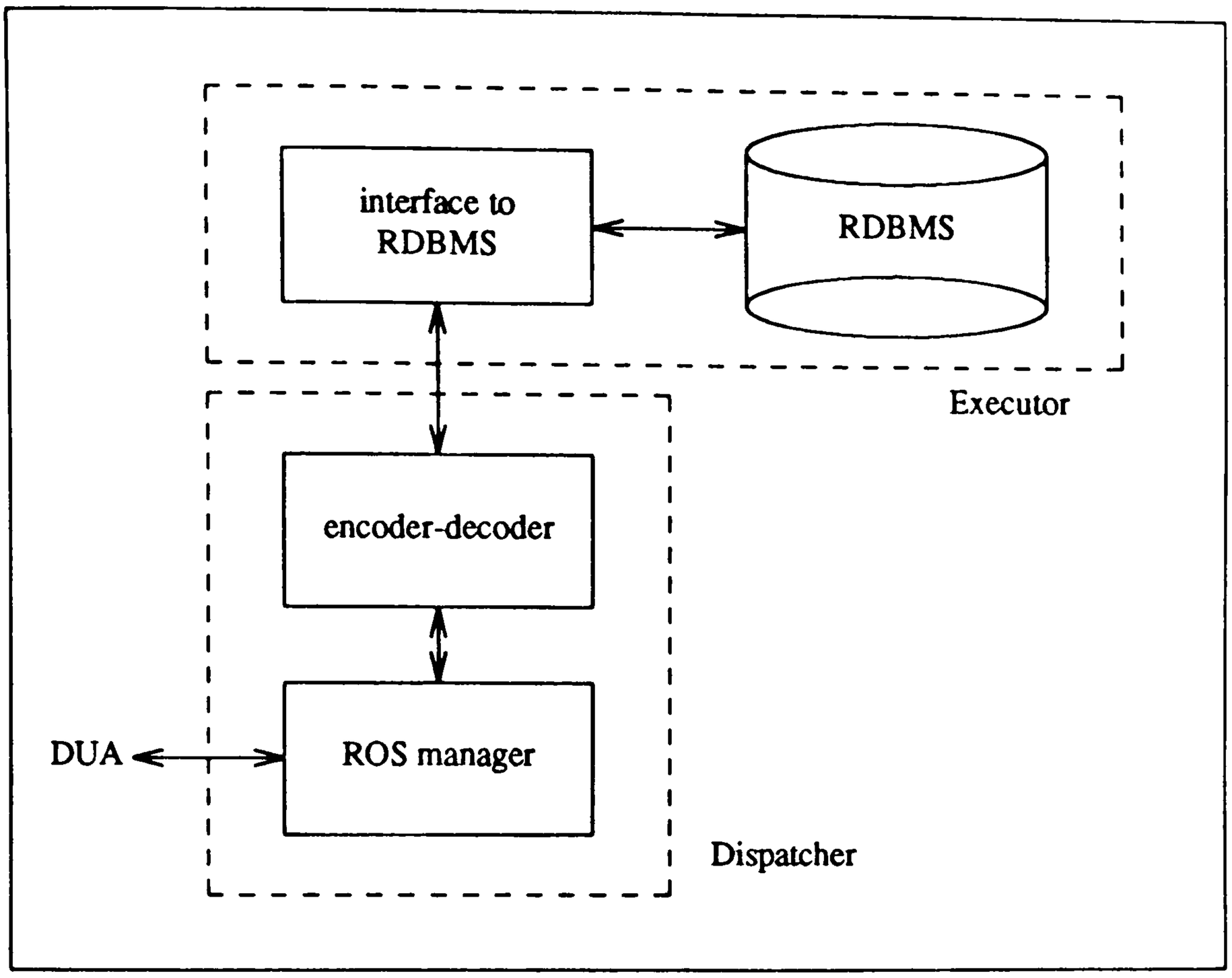


Figure 7.1: Functional architecture of the DSA

The executor

The executor executes operations on local information. This information is stored by the RTI Ingres RDBMS and the executor must map from C data structures and functions into relational queries, expressed in terms of the QUEL query language. This is the task of the interface element which is written using the embedded EQUEL/C programming interface.

The executor therefore performs the mapping from the directory local conceptual model to a specific local internal model which is, in fact, the relational data model. It is also responsible for implementing local transaction control, access controls and integrity checking.

7.2.2. Functional architecture of the DUA

The structure of the DUA may also be divided into two functional elements as shown by figure 7.2 below.

- The *accessor* [1] is the counterpart of the DSA dispatcher. It manages associations with DSAs and requests operations via the Remote Operations Service.
- The *application interface* presents an interface to a directory user (human or application).

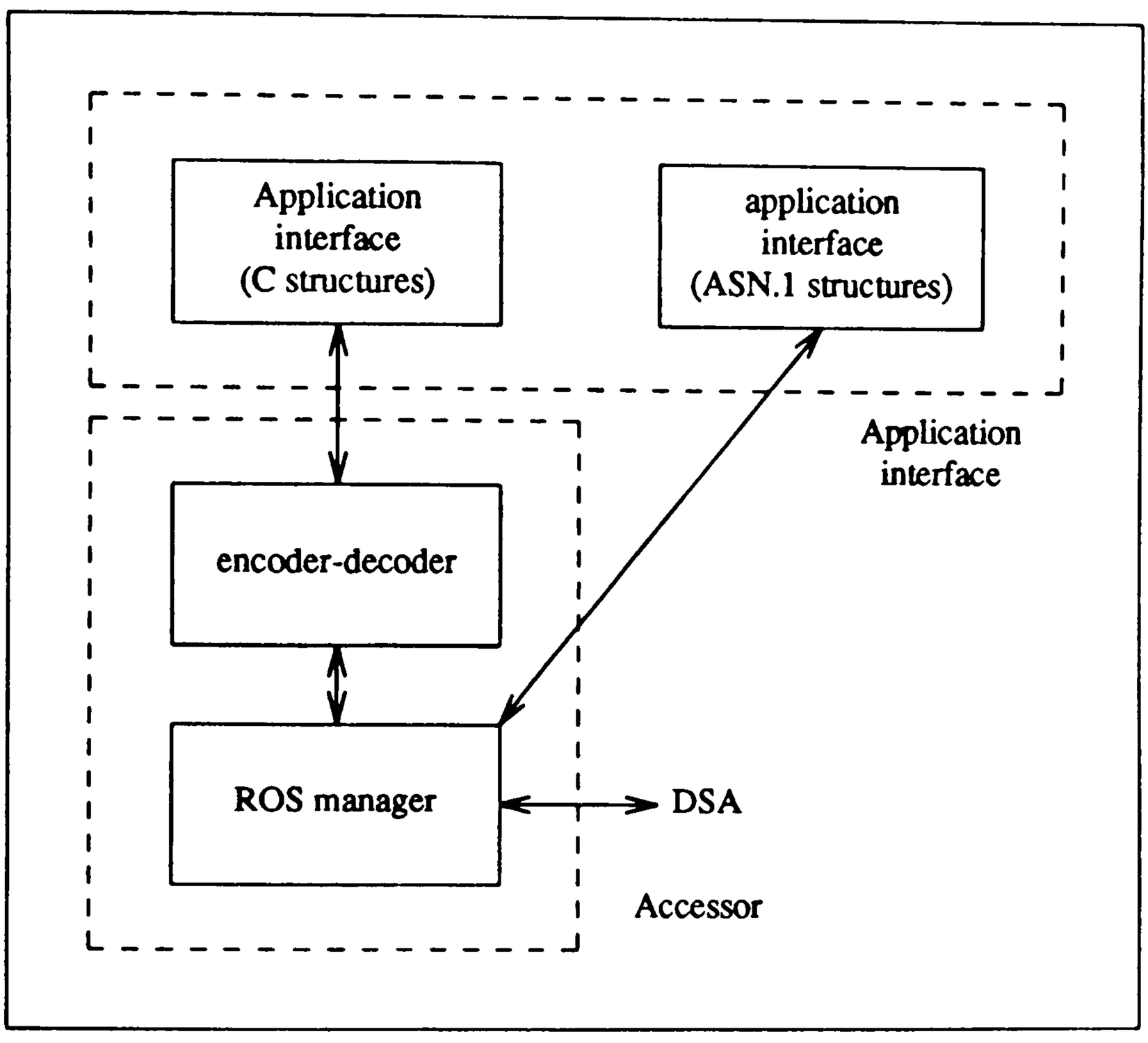


Figure 7.2: Functional architecture of the DUA

The accessor

The accessor is common to all DUAs and is implemented by a standard DUA library. This library provides a number of routines concerned with establishing and maintaining associations with specified DSAs. Operations may be requested via a ROS connection and applications may deal directly with ASN.1 arguments and results. Alternatively, the accessor provides an encoder/decoder used to map between C data structures and ASN.1 data structures.

The application interface

Each application accessing the Directory implements its own application interface. The application interface maps between the application's external model and the directory global conceptual model. Application interfaces may vary in structure from libraries of routines, included within other applications, to sophisticated interfaces interacting directly with humans. Section 7.4 will outline several simple application interfaces.

7.3. Implementation of the DSA

The previous section specified the functional architecture of the DSA. This section examines the DSA implementation in greater detail.

- Section 7.3.1 describes the overall design of the DSA in terms of the modules implementing specific DSA functions.
- Section 7.3.2 describes the relational model representing directory information, access controls and definitions.
- Section 7.3.3 describes the implementation of the navigation procedure and the distributed Read Entry operation.

7.3.1. Modules and DSA functions

The functionality of the DSA is implemented by a set of program modules, each of which is responsible for a different aspect of its operation. These modules and their interactions are shown in figure 7.3 and are described by the following paragraphs.

Dispatcher modules

The dispatcher part of the DSA is implemented by the *ROS manager*, *encoder-decoder*, *navigation* and *operation select* modules.

The *ROS manager* contains routines to accept and terminate associations from DUAs and other DSAs. Each new association creates an entry in the *associations table* containing an association identifier and indicating the reference of the user's home DSA. The ROS manager module also receives operations, results and errors via the Remote Operations Service and is responsible for ensuring that these are handled correctly.

Each new operation request is passed to the *operation select* module, responsible for overseeing its general execution. This module decodes arguments from ASN.1 into C data structures and passes the operation to the relevant module in the executor. Results and errors are

encoded back into ASN.1 data structures and returned to the ROS manager. Encoding and decoding uses routines from the *encoder-decoder* module.

Read Entry operations may require distributed navigation. In this case, the operation select module passes the operation to the *navigation* module which determines the reference of a closer DSA to the target entry. If this reference indicates the local DSA, the operation is passed to the executor as before. Otherwise, the ROS manager is instructed to return a referral or to open an association with the remote DSA and chain the operation. The choice of action depends upon whether the chaining or referral mode of operation is specified within the operation header.

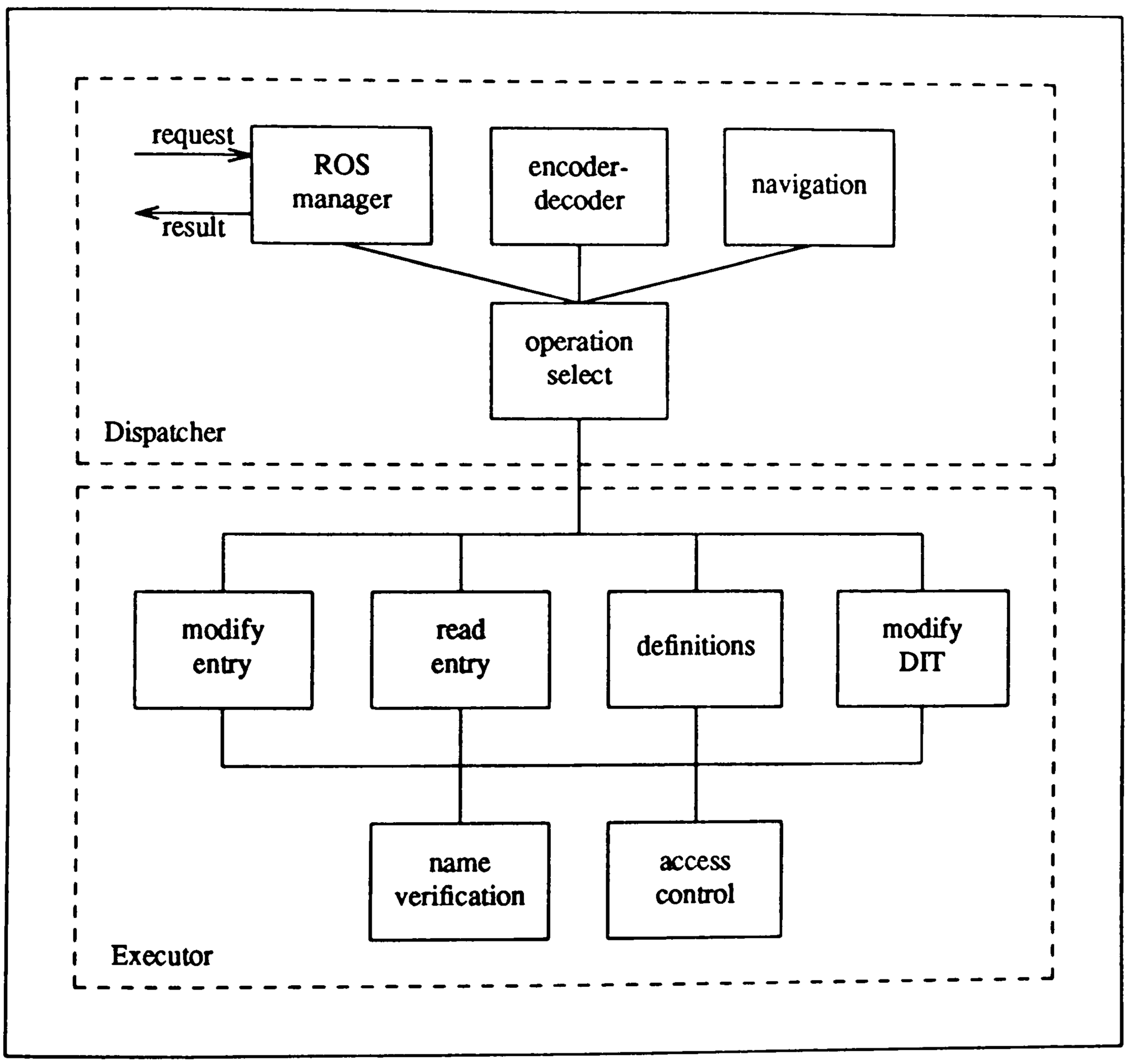


Figure 7.3: Program modules implementing the DSA

Executor modules

The executor contains those modules executing directory operations on the Ingres database as well as the *name verification* and *access control* modules.

- The *read entry* module implements the Read Entry operation and contains routines to retrieve attributes, access controls and definitions.
- The *modify entry* module contains those routines implementing the Modify Entry operation.
- The *modify DIT* module is responsible for the *Add Entry*, *Delete Entry*, *Add Alias* and *Delete Alias* operations.
- The *definitions* module contains routines to manipulate and retrieve entry and attribute definitions.

Each of these modules performs name verification and access control. This is achieved via routines from the *name verification* and *access control* modules respectively. If successful, the arguments for the requested operation are mapped into an EQUEL query which is executed by the Ingres RDBMS. The results of this query are then mapped back to a C data structure which is returned to the operation select module within the dispatcher.

The operation of the DSA is demonstrated by the following example. Consider the execution of a *Modify Entry* operation. The operation request is received by the ROS manager on an open association. It it passed to the operation select module which decodes its arguments using the encoder-decoder module. Operation select then passes it to the modify entry module where name verification is performed and access controls are checked via calls to the relevant modules. If successful, the operation is executed on the Ingres database and the results returned to the operation select module. Here they are encoded into ASN.1 and returned to the calling DUA via the ROS manager module.

This section has outlined the general execution of a single operation in terms of its progress through a set of DSA program modules. In reality, the Directory will allow many DUAs to simultaneously request operations at each DSA. Consequently, DSAs must be capable of servicing multiple, asynchronous connections. If this were not the case (e.g. DSAs were *single threaded*), the possibility of deadlock between DSAs would arise. For example, two DSAs might simultaneously chain an operation to each other during navigation. If these DSAs were single threaded, they would wait for each other to process the operation before continuing, resulting in deadlock.

This concludes the general description of the DSA implementation in terms its program modules. The following section considers the operation of the executor modules in greater

detail. In particular, it describes the representation of directory information within the Ingres RDBMS.

7.3.2. The relational model representing directory information

The RTI Ingres relational database provides the storage mechanism for the DSA implementation. This requires a mapping from the directory information model to the relational data model.

Several aspects of this mapping are of particular interest:

- A relational representation of entries, attributes and the Directory Information Tree.
- Implementing name verification.
- Implementing operations and local transaction control.
- A relational representation of access controls.
- A relational representation of entry and attribute definitions.

The following sections describe the Ingres relations implementing each of the above areas. It should be noted that, on the whole, directory operations are implemented by direct access to information within the Ingres database and not by loading information into dynamic memory and manipulating it there (the approach of some existing systems such as the *BIND* nameserver). This solution enables the DSA to utilise features of the Ingres database such as transaction control.

Representing entries, attributes and the DIT.

The Directory Information Tree describes the fundamental structure of directory information. The DSA implementation maps the DIT and its contents to two relations.

- The *distinguished names* relation stores directory names and aliases. It also specifies the parent-child relationships between entries.
- The *attributes* relation stores the attributes belonging to each entry.

Figure 7.4 shows the structures of these two relations. Throughout the following work, the primary keys of relations appear in bold type and the labels *(int)* and *(string)* indicate whether fields have integer or character string values.

The *distinguished names* relation					
entry id (int)	coded name (string)	superior id (int)	alias for (int)	ACL id (int)	entry def id (int)
1	/C=GB/	0 (root)			1
4	/C=GB/O=Nott.Uni/	1			3
7	/C=GB/O=Nott.Uni/OU=CS/	4		2	4
11	/C=GB/O=Nottingham/	1	4		

The *attributes* relation			
entry id (int)	attribute def id (int)	**type** (string)	value (string)
4	6	phone	44 602 484848
4	9	location	Nottingham
7	6	phone	44 602 484848 2862
7	6	phone	44 602 484848 3334
7	9	location	Tower Block

Figure 7.4: The structure of relations representing the DIT and entries

Each directory information structure (i.e. each entry, ACL, entry definition etc.) is assigned an integer reference forming its unique system wide identifier or key. Thus, entries are identified by *entry ids*, access control lists by *ACL ids* and entry and attribute definitions by *entry def ids* and *attribute def ids* respectively.

A tuple in the distinguished names relation stores the general information describing a single directory entry. This information is required for name verification and alias resolution as described in the following section.

The *entry id* field identifies the entry in question and the *superior id* field specifies its parent in the DIT. This supports the *List Subordinates* operation which retrieves all tuples with a specified superior identifier. The distinguished name of the entry is encoded as a character string and stored in the *coded name* field. This encoding has been used to represent names as character strings throughout this thesis and was specified in section 3.2.

The *alias for* field indicates whether this tuple represents an alias and, if so, contains the identifier of the aliased entry. Otherwise it is empty. For example, the entry with id 11 in figure 7.4 is an alias for the entry */C=GB/O=Nott.Uni/* with id 4.

Finally, the *ACL id* and *entry def id* fields identify the entry level ACL and the entry definition constraining this entry. They point to tuples in the *ACLs* and *entry defs* relations as described below.

Each tuple in the attributes relation represents a single attribute value belonging to a specific entry. The *entry id* field identifies the entry containing each attribute and the *attribute def id* field specifies the attribute definition governing this attribute. The *type* and *value* fields store the attribute type and value. Figure 7.4 shows example attributes belonging to the entries */C=GB /O=Nott.Uni/* (entry id = 4) and */C=GB /O=Nott.Uni /OU=CS/* (entry id = 7).

Name verification

Name verification requires that the DSA maps from a purported name to an entry in the DIT. More precisely, the purported name is mapped into an entry id. The DSA performs name verification by the following steps:

1. The attributes of the purported name are encoded as a character string (see above).
2. The *distinguished names* relation is searched for tuples whose *coded name* field is a prefix of this string. This requires a QUEL *retrieve* query.
3. If no match is found, a name error is generated.
4. Otherwise, the tuple matching the greatest number of name parts is selected.
5. If this tuple represents an alias, the coded name of the aliased entry is retrieved by a second QUEL query. This replaces the matching alias in the purported name and the verification process is started again.
6. If the best match is not an alias and matches the entire purported name, the target entry has been located and its entry id is returned.
7. If the best match is a "proper" prefix of the purported name, no exact match was found and a name error is therefore generated.

Step 2 indicates that name verification does not perform a tree walk on the DIT as one might expect. Instead, the local name space is treated as flat and is subject to an exhaustive search. This is considered more efficient for small volumes of data because Ingres queries are the most time consuming aspect of DSA operation and the above method reduces to one query provided that aliases are not encountered. This is quicker than a tree walk generating a query at each node. Furthermore, the use of indexing techniques such as *B-Trees* and *hashing* facilitates efficient searching. This name verification technique might require revision for production systems storing large volumes of information. However, it is sufficient for the storage of relatively small volumes of information (less than 500 entries) during piloting.

Implementing operations and transaction control

This section discusses the mapping of directory operations to relational queries and the implementation of the one out - all out transaction control policy.

Directory operations are mapped into QUEL statements which are executed on the relations storing directory information. Those operations reading, browsing and manipulating the Directory Information Tree access the *distinguished names* and *attributes* relations described above. Operations manipulating access controls and definitions access a number of additional relations which are specified later.

Each directory operation may generate a number of QUEL statements accessing several relations using the *retrieve*, *append*, *replace* and *delete* expressions. For example, an *add entry* operation might append one tuple to the *distinguished names* relation and several tuples to the *attributes* relation. It might also retrieve tuples from relations representing ACLs and definitions. The combination of these QUEL statments should form an atomic transaction obeying the directory *one out - all out* transaction control policy. A transaction control mechanism is therefore required to ensure that concurrent QUEL queries do not conflict. This mechanism has been implemented using the Ingres transaction control mechanism allowing application programs to indicate the start of transactions and to specify savepoints, abort and commit statements. Thus, each directory operation generates several QUEL queries surrounded by transaction control statements.

This chapter does not describe the EQUEL/C code implementing directory operations in detail. However, the following pseudo-code should give the flavour of this mapping and indicates the use of Ingres transaction control statements.*

```
        /* Modify the attributes belonging to an identified entry */
##      begin transaction

        /* add any values */
        for (i in new attributes) {
##              append to attributes (entry_id = id, attribute_def_id = def, type = types[i], value = new_values[i])

                if (update failed) {
##                      abort
                        return(ERROR);
                }
        }

        /* replace any values */
        for (i in replacement attributes) {
```

---

*## characters indicate that the following line of code is an EQUEL statement. Ingres relation and field names are shown in italics.

```
##              replace attributes (value = new_values[i]) where entry_id = id and type = types[i]

                if (update failed) {
##                      abort
                        return(ERROR);
                }
        }

        ...
        /* successful end of modifications */
##      end transaction
        return(OK);
```

Representing access controls

Chapter 4 specified a directory access control mechanism based on Access Control Lists. This section describes the representation of ACLs within the supporting relational database. In particular, it considers:

- A relational representation of access categories.
- A relational representation of Object Set Descriptors, including subtrees and filters.
- Associating ACLs with specific entries and attributes.

Figure 7.5 indicates the structure of the *ACLs* and *type ACLs* relations solving these problems.

The *ACLs* relation				
ACL id (int)	categories (int)	base entry (string)	level (int)	filter (string)
1	0147	/C=GB/O=Nott.Uni/		(title=manager)
1	0101	/C=GB/O=Nott.Uni/	2	(title=manager)v(office=102)

The *type ACLs* relation		
entry id (int)	**type** (string)	**ACL id** (int)
4	phone	1
7	phone	2

Figure 7.5: The structure of relations representing access controls

Each tuple in the *ACLs* relation represents a single ACL Element. Thus, an Access Control List may be represented by a set of several tuples. The *ACL id* field identifies the ACL to which each element belongs and the *categories* field stores the bitstring representing the access categories granted by this element. The remaining fields represent an Object Set Descriptor in terms of its *base entry* (root of subtree), *level* indicating the depth of the sub-tree and a *filter* represented by a character string encoding. This encoding uses nested parentheses to represent nested sub-filters and the *n*,*v* and ¯ symbols to represent the logical *and*, *or* and *not* operators between sub-filters. These are combined in an infix notation. For example, *((title=manager) v (office=102)) n (class=person).*

The DSA contains routines to map from filter structures to their encodings and back again. The use of a character string encoding for filters provides an economic storage technique for piloting although it is not clear how this method can be extended to support different attribute syntaxes.

The *type ACLs* relation represents the binding between entries, attributes and attribute level ACLs. Each tuple of this relation identifies the ACL controlling access to the named attribute type within the referenced entry.

Checking access controls requires the following queries to the database.

- The *ACL id* of the relevant Access Control List is retrieved from the *distinguished names* relation (for entry level ACLs) or from the *type ACLs* relation (for attribute level ACLs).
- The *ACL id* is used to retrieve all tuples representing the ACL from the *ACLs* relation.
- These tuples are searched for those whose *categories* field assigns the required access rights.
- Checking of subtree membership and filters is achieved by comparisons of user information against the retrieved tuples.

Representing entry and attribute definitions

Entry and attribute definitions are mapped into several relations within the DSA's internal model. These are shown in figure 7.6 below.

The *attribute defs* relation			
attribute def id (int)	type (string)	scope (string)	recurring (int)
6	phone	/C=GB/	1 (true)
9	location	/C=GB/	0 (false)

The *entry defs* relation						
entry def id (int)	type (string)	scope (string)	contains (int)	RDN attribute (int)	default ACL (int)	ACL (int)
4	organisation	/C=GB/		2		
7	org unit	/C=GB/		2		

The *superiors/mandatory/ optional* relations	
id (int)	**associated id** (int)

Figure 7.6: The structure of relations representing entry and attribute definitions

Attribute definitions are represented by the *attribute defs* relation. Each tuple of this relation assigns a unique identifier to an attribute definition, stores its type, scope and an indication of whether it may have recurring attribute values. The scope is represented by an encoded distinguished name as described above.

Entry definitions are represented by the *entry defs*, *superiors*, *optional* and *mandatory* relations. Each tuple of the *entry defs* relation represents a single entry definition. It assigns a unique identifier to the definition, describes its type and scope and contains the identifiers of any nested entry definitions. In addition, it specifies the naming attribute for entries of this class and indicates the ACLs affecting this definition.

The *superiors*, *optional* and *mandatory* relations associate specific entry definitions with superior entry definitions or with mandatory and optional attribute definitions. Each tuple of these relations associates an identified definition with another entry or attribute definition.

Although entry and attribute definitions are maintained within the Ingres relational database, the prototype DSA also supports the caching of definitions in memory. Whenever a new definition is retrieved from the supporting database, it is retained in memory for later use. This is sensible due to the high rate of access to definitions during operations such as *modify entry* and due to the low rate of updates to definitions. The addition and deletion of definitions accesses the relations described above and also forces updates to the cache.

The previous paragraphs have described the mapping from the directory information model to the relational data model and have presented a brief overview of the use of the RTI Ingres database in constructing the prototype DSA. The following section turns its attention to the issue of navigation and storing minimal and opportune knowledge within the pilot DSA.

7.3.3. Implementing knowledge and navigation

The DSA implementation exhibits a limited capability for distribution. This takes the form of a distributed Read Entry operation supported by the DSA navigation procedure. The replication of information between DSAs is not supported by the prototype.

The prototype DSA uses both minimal and opportune knowledge during navigation and this is stored within several Ingres relations. As with entry and attribute definitions, it is beneficial to maintain a dynamic, in memory cache of knowledge in order to speed up the navigation process. This is sensible due to the relatively small amount of knowledge information requiring storage and due to the low frequency of knowledge updates. The knowledge cache is initialised when the DSA is booted and may be altered following corrections to knowledge inconsistencies.

The prototype DSA supports both the chaining and DSA referral modes of operation and is able to detect and correct knowledge inconsistencies via the hop-count, intended reference and route checking mechanisms. The routines to perform knowledge error detection and correction belong to the *navigation* module and are called by the *operation select* module before an operation is passed to the executor part of the DSA.

The distributed Read Entry operation supports distributed access control. This utilises implementations of the *Check Filter* and *Retrieve Reference* operations belonging to the Directory System Protocol. These operations also require navigation and routing.

This concludes the description of the DSA implementation.

7.4. Implementation of DUAs

Several Directory User Agents have been developed during prototyping of the Directory service. These have tested the DSA implementation and have demonstrated possible uses of the Directory in supporting other distributed applications. This section outlines the construction of three pilot DUAs and demonstrates how they are supported by the basic *libdua* library. The first of these is a simple interactive DUA, used to test the DSA implementation and to facilitate the input and modification of directory information. The other DUAs have been developed within the framework of the *AMIGO MHS+* project with the intention of supporting an experimental distribution list protocol. They allow the expansion of distribution lists by an automated *group agent* and also the remote query of the Directory service using X.400 as a transport medium for directory operations.

This section is structured in the following way.

- Section 7.4.1 describes the *libdua* library supporting all three prototype DUAs.
- Section 7.4.2 briefly outlines the construction of the interactive DUA.
- Section 7.4.3 summarises the AMIGO distribution list protocol and proceeds to discuss the implementation of two DUAs supporting the expansion and maintenance of distribution lists.

7.4.1. The libdua library

The *libdua* C library provides basic support for Directory User Agents. It implements the *accessor* element of the DUA (see figure 7.2) and is therefore responsible for managing associations with DSAs, requesting directory operations and encoding and decoding their arguments and results.

Association management is achieved by the *ROS_init*, *ROS_release* and *ROS_call* routines.

- *ROS_init* establishes a connection with a named DSA and initialises the Remote Operations Service. The target DSA is identified in terms of an *application entity title* and a *host name* (e.g. <*cs-dsa*, *sun1*>). This routine returns an integer association identifier referring to the association.
- The *ROS_release* routine terminates an identified association.
- *ROS_call* requests a remote operation over an established connection. Arguments, results and errors are passed to and from this routine as ASN.1 data structures.

A directory user may supply their distinguished name to ROS_init for authentication purposes. However, a password scheme has not yet been implemented. Alternatively, the user

may request "public" directory access. This mechanism is supported in order to test the directory access control mechanism.

These routines draw heavily on various ISODE libraries and require that the application interface deals with ASN.1 data structures. The libdua library also supports the use of C data structures via the *encoder-decoder* routines. Each argument, result and error type has a corresponding routine to map from C to ASN.1 or vice versa. In addition, a high level controlling routine is implemented for each directory operation. This takes C arguments, calls the relevant coding routines, passes the operation to *ROS_call* and then decodes and returns the results. The combination of coding and controlling routines frees interface designers from dealing directly with ASN.1. Figure 7.7 shows an example use of the libdua routines when requesting a Read Entry operation from within an application interface.

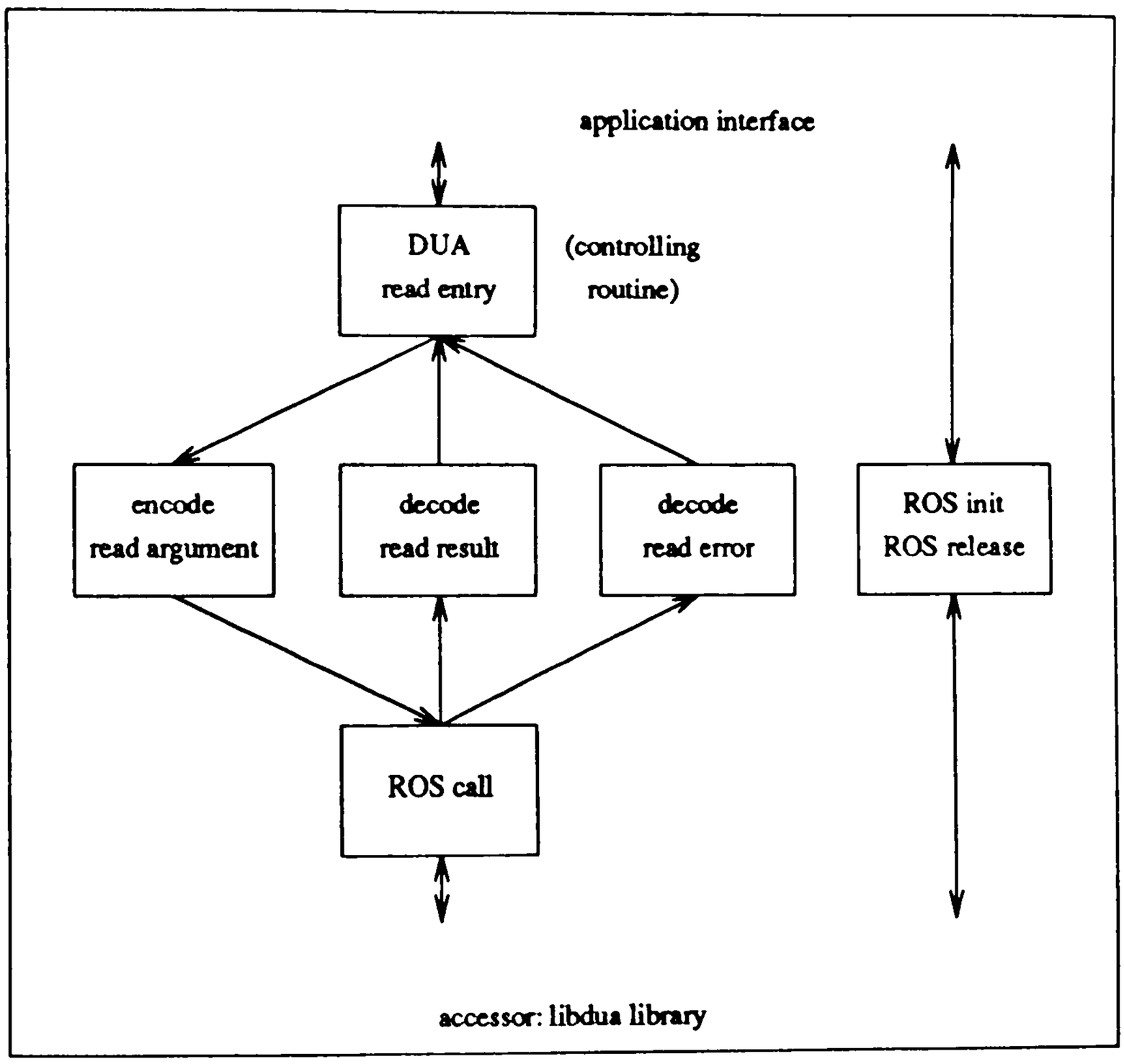


Figure 7.7: Libdua routines used in a Read Entry operation

The remainder of this section demonstrates the use of the libdua library in implementing several different DUAs.

7.4.2. An Interactive Directory User Agent

The primary motivation for implementing the interactive DUA has been the testing of the prototype DSA. This DUA allows the user to connect to a named DSA and to interactively perform a series of ad-hoc operations.

The user interface is extremely simple, consisting of a top level menu to select the desired operation followed by prompts for operation arguments. Results are printed to the screen. Although the interface is simple, the interactive DUA does support most operations belonging to the Directory Access Protocol. It allows the retrieval of information, the browsing of the DIT and operations to update entries and attributes. It also supports the management and retrieval of access controls and entry and attribute definitions.

The user may request authenticated or public access at the start of an association and may also select whether the chaining or hints mode is to be used during distributed Read Entry operations.

The design of a sophisticated user interface has not been a goal of this research. Consequently, the interactive DUA will not be described in further detail.

7.4.3. The AMIGO MHS+ Directory User Agents

AMIGO MHS+ was a European Community *COST-11-TER* funded research project studying *advanced messaging* and *group communication* [SPET88]. The author's involvement with this project included the specification and pilot implementation of a distribution list protocol based on X.400 Message Handling Services (MHS).

The AMIGO distribution list protocol uses MHS for message transfer, the Directory service for storing list information and a special application called the *group agent* for the expansion of distribution lists.

The following are the major features of this protocol.

- Distribution lists have complex functionality supporting *moderating*, *auditing* and *administrative* functions.
- The *group agent* belongs to the X.400 P2 layer and accesses the Directory service for information concerning distribution lists.
- Humans may also access the Directory for distribution list information. Due to the lack of a commonly available Remote Operations Service, this uses X.400 as a transport system for directory queries and results.

A full specification of the AMIGO distribution list protocol was published in [BENF87] and [BENF88a]. The following paragraphs present a brief summary of this work. They then

proceed to discuss the representation of distribution lists within the Directory service and the implementation of two DUAs providing group agent and human access to the Directory.

The structure of a distribution list

In its abstract form, a distribution list is structured as a set of properties and operations. The properties of a distribution list are described by the following table.

Properties of an AMIGO distribution list	
property	meaning
Name	The X.400 Originator/Recipient address of the list
Description	Text describing the purpose of the list
Charging Algorithm	Text describing the list accounting procedure
Joining Procedure	Text describing how to join the list
Submitting Members	Entities who may send messages via the list
Receiving Members	Entities who receive messages from the list
Auditor	The entity who receives (non)delivery notifications
Moderator	The entity who receives unauthorised contributions
Administrator	Entities who may modify the properties of the list
Reply To	The default reply to users for the list

These properties may be retrieved and updated by the following conceptual operations.

Operations on an AMIGO distribution list	
operation	meaning
Retrieve Members	Returns the names of submitting/receiving members
Verify Member	Determines if named entity is submitting/receiving member
Describe List	Returns a set of requested properties
Add Member	Adds a new submitting or receiving member
Delete Member	Deletes an existing submitting or receiving member
Modify Property	Modifies some of the above properties

These operations are subject to access controls. This combination of properties and operations is one example of an application specific directory external model.

Expansion of distribution lists

This section summarises the operation of the group agent when expanding messages sent to a distribution list and transmitting them to the list recipients. Figure 7.8 shows the entities involved in the operation of the AMIGO distribution list protocol.

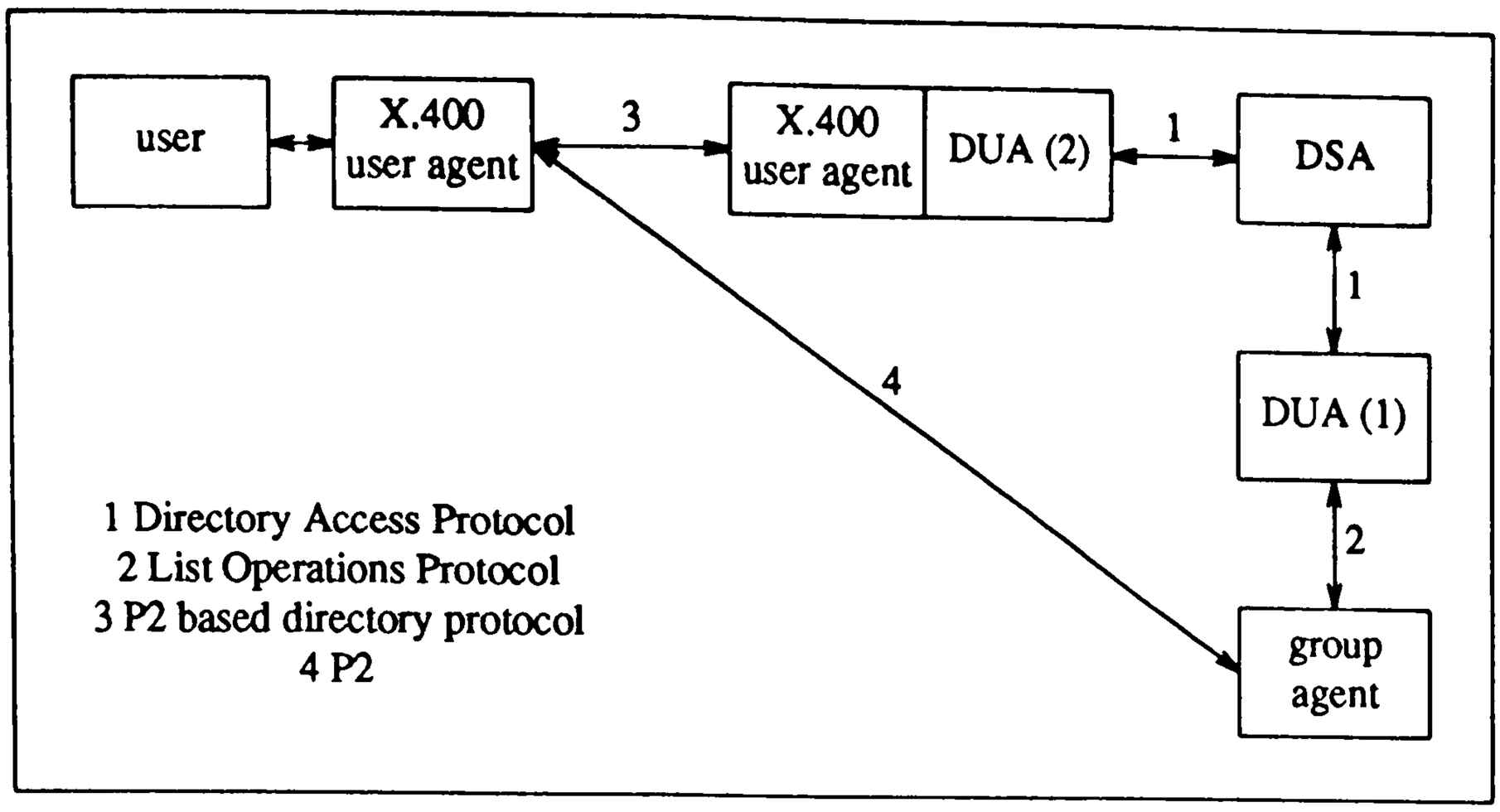


Figure 7.8: Functional model of the AMIGO distribution list mechanism

List expansion proceeds as follows.

- The user sends a message to the group agent via the X.400 MHS (protocol 4).
- The group agent interacts with the Directory service to determine whether the sender is a valid submitting member (protocol 2).
- If so, the group agent retrieves the names of the receiving members, the reply-to users and the list auditor from the Directory and retransmits the message to the receiving members.
- If not, the group agent retrieves the name of the list moderator from the Directory and sends the message there.

A user may also query the Directory concerning the properties of a distribution list. This proceeds as follows.

- The user expresses their query in terms of the list operations described above. These form a human readable body part of an X.400 IP-message (protocol 3).
- The X.400 encoded query is sent to a special User Agent/DUA combination via the MHS.
- The Directory User Agent parses the request and calls the relevant directory operations (protocol 1).

- The results are wrapped in an X.400 IP-message and returned to the user (protocol 3).

Both the group agent and user interactions with the Directory service require the support of special Directory User Agents. These are described below.

The group agent DUA

The group agent interacts with the Directory service during the expansion of messages. The group agent therefore contains a DUA (DUA 1 in figure 7.8) supporting its external model in terms of the properties and operations described above. The application part of the *group agent DUA* maps this external model into directory operations, entries and attributes. This includes:

- Mapping each distribution list to a single directory entry.
- Mapping distribution list properties to directory attributes.
- Mapping distribution list operations to directory operations. This mapping is shown by the table below.

Mapping list operations to directory operations	
list operation (external model)	directory operation (conceptual model)
Retrieve Members	Read Entry
Verify Member	Read Entry
Describe List	Read Entry
Add Member	Modify Entry
Delete Member	Modify Entry
Modify Property	Modify Entry

Operations are subject to access control and the group agent must therefore be authenticated by the Directory service. It follows that the group agent must be named within the Directory Information Tree.

Support for the AMIGO distribution list protocol requires the creation of a new entry definition for the *distribution list* object class. It also requires several new attribute definitions specifying distribution list properties.

The application part of the group DUA consists of the *groupdua* library containing routines mapping from distribution list operations to the controlling routines supported by the *libdua* library. The group agent therefore contains both the *libdua* and *groupdua* libraries and can call list operations as required.

The X.400 access DUA

The AMIGO protocol supports human access to distribution list information by embedding queries into X.400 IP-messages which are transmitted to the Directory service. This use of X.400 as a transport service for operations and results facilitates remote access by users at sites not supporting the Remote Operations Service. Operations are written in a simple human readable text format closely resembling the list operations described above. They arrive at a special user agent which calls a DUA (DUA 2 in figure 7.8) to map them into directory operations.

This DUA is implemented by a YACC [JOHN78] based parser which reads a text query and constructs a number of C operation structures. These are then passed to the controlling routines in the accessor.

The pilot implementation uses the PP Message Handling System [KIL88a, ONIO87] to provide the X.400 service. Thus, the user agent and parser can be included within a special PP *channel*.

This concludes the discussion of DUA implementations. The final section of this chapter summarises prototyping activities and presents the insights gained as a result of implementation work.

7.5. Summary and conclusions of implementation work

This chapter has described the implementation of a prototype Directory service supporting many of the concepts developed by previous chapters.

The prototype consists of a DSA, supporting a local Directory Access Protocol and distributed Read Entry operation, as well as several DUAs, supporting both human and application uses of the Directory.

The DSA utilises the *ISO Development Environment* (ISODE) and *RTI Ingres RDBMS* software tools to manage communication and data storage respectively. The bulk of the DSA is written in the C programming language, requiring a mapping from ASN.1 data structures to C data structures. This is achieved using the ISODE *PEPY* parser/generator. Use of Ingres requires a mapping from the directory information model to the relational information model. An important feature of this work is the use of the Ingres transaction control mechanism to implement *one out - all out* transaction control within the DSA.

Three DUAs have been implemented during prototyping activities. The interactive DUA generates ad-hoc queries to test the operation of DSAs and to load and modify directory information. The remaining DUAs have been implemented to test the *AMIGO MHS+* distribution list protocol. The first of these supports a special X.400 user agent called the *group agent* in its run time handling of messages sent to distribution lists. The second allows remote users to query the Directory service for distribution list information using X.400 MHS as a transport medium for text-encoded queries.

Section 7.1 postulated three motivations for implementing the prototype Directory service.

1. Testing and refining the results of specification work.
2. Exploring the use of existing software tools in supporting Directory services.
3. Indicating issues to be addressed by production Directory services.

The following sections explain how these motivations have been satisfied.

7.5.1. Testing and refining the directory model

Implementation work occurred in parallel with specification work and formed an integral part of the design process. Prototyping has been instrumental in revealing many problems and shortcomings with earlier concepts. For example, prototyping indicated the requirements for transaction control, a directory superuser and knowledge management facilities.

In addition, the prototype Directory indicates that concepts specified by earlier chapters are implementable. This is particularly true of the access control and integrity mechanisms which are both supported by the prototype DSA.

7.5.2. Use of software tools

Implementation has tested the suitability of the *ISODE* and *RTI Ingres* products for supporting Directory services.

ISODE has proved to be an essential tool for the development of the prototype. Much implementation work concerned the management of remote operations and parsing and generating ASN.1 data structures. The implementation of routines to manage ASN.1 to C conversion would have been time consuming and repetitive without the PEPY program. Furthermore, this task is likely to be repeated during the development of many other services. One can conclude that the widespread availability of tools such as ISODE will be vital to the future development of distributed applications.

It is interesting to note that, during the course of this research, ISODE has expanded to include the *ROSY* and *POSY* [ONIO88] tools. These support the automatic generation of stub routines from abstract operation definitions and the automatic generation of C structures

from ASN.1 code. These tools would also have proved extremely useful during implementation work.

The RTI Ingres RDBMS encouraged the high speed development of the prototype. The simplicity and flexibility of the relational model supported changes to the directory information model with minimal recoding. Furthermore, features such as the transaction control and crash recovery mechanisms proved extremely useful. On the other hand, the relational model is quite dissimilar in structure to the Directory Information Tree and the mapping between the two is not always natural. Consequently, mapping directory operations to relational queries imposes a substantial overhead on the DSA.

In conclusion, Ingres provided an excellent prototyping tool although it might prove less suitable for use within production systems.

7.5.3. Implications for production Directory services

The implementation and testing of the prototype Directory service has revealed a number of issues to be addressed by production systems.

- Direct interaction with the Directory was often difficult, even when accessing small volumes of data. The raw Directory Access Protocol is complex and confusing and, in particular, it was hard to remember names, the structure of the tree and the contents of entries. This implies that the development of sophisticated user interfaces will be essential to support human Directory use.
- The prototype has shown a clear need for system management tools to support the configuration of new DSAs. An important aspect of this is the ability to load large volumes of initial information from sources such as UNIX *password* files.
- Representing AMIGO distribution lists within the Directory service required the creation of several new entry and attribute definitions. This reinforces the requirement for the Directory to allow the dynamic definition of integrity rules in order to support non-standardised activities.
- Finally, the poor distributed performance of the prototype suggests that production Directories must implement caching techniques and replication strategies in order to obtain an acceptable response time.

Chapter 8

Conclusions and Future Directions

This thesis has described the specification and implementation of a distributed Directory service supporting computer based communication and distributed applications. In particular, it has considered the use of the Directory service for the management of communication information and has explored a number of management related issues. This work has also considered the distributed operation and management of the Directory service in detail.

This chapter presents the major conclusions of this work and examines possible directions for further research in this area.

The completion of this thesis is particularly relevant at the present time due to the impending publication of the first (1988) ISO/CCITT *X.500* international standard for Directory services and the subsequent start of the 1988-1992 study period. Consequently, a section of this chapter is devoted to exploring the implications of this work for the future development of the standard.

This chapter is structured in the following way.

- Section 8.1 restates the motivations and goals which have driven this research.
- Section 8.2 presents conclusions for information management within the Directory service.
- Section 8.3 presents conclusions for the distributed operation of the Directory service and, in particular, the management of knowledge and replication.
- Section 8.4 presents conclusions arising from the implementation of the prototype Directory service.
- Section 8.5 describes the possible implications of this research for the X.500 standard.
- Section 8.6 outlines a number of unresolved issues, not addressed by this thesis, warranting further immediate study.
- Section 8.7 discusses the future role of Directory services and outlines a number of issues for longer term research.

8.1. Goals and motivations of this research

The Directory service provides vital support for electronic communication and distributed applications running over computer networks. In doing this it plays several roles:

- It supports a unified namespace, independent of different lower level application address spaces.
- It allows both humans and applications to retrieve the information required for communication.
- It manages communication information shared by different distributed applications.

At the inception of this research the following observations applied to existing nameservers and Directory standardisation efforts. They are still largely true today.

- The issues of naming and information retrieval have been addressed by existing nameservers such as *BIND* and the *Clearinghouse*. These systems have, in turn, provided input for the X.500 standard. Although this work requires refinement, the basic concepts are well established.
- On the whole, the issue of managing communication information has been neglected. Most communication services maintain their own information bases with varying degrees of success (e.g. electronic mail routing tables). It is unlikely that this approach will cope with the expected rise in the number of services and service users in the near future. Consequently, the Directory service must provide support for the management of communication information shared between many different services.
- The global Directory will be provided by an interconnection of heterogeneous local Directories whose configuration will change over time. The consistent management of knowledge and other configuration information will be vital to its successful operation. However, the management of existing systems is currently achieved by ad-hoc methods and there is little formal support for system reconfiguration. Furthermore, there is little discussion of Directory system management within the X.500 standard.
- The use of replicated information may increase the efficiency and robustness of directory operation. Replication requires mechanisms to establish and manage agreements between local Directories and to ensure the consistent update of replicated information. Current systems provide little support for replication.

These observations provided the motivation for much of this thesis and the major goals of this research are summarised below.

- Supporting information management within the Directory service.
- Supporting the management of the distributed Directory service itself.

Information management was considered in chapter 4 which specified the functionality to ensure that updates to information are both legal and valid and also reflect real world management policies. The conclusions to this work are presented in section 8.2 below.

The management of the Directory service was considered as part of chapters 5 and 6. This work concerned the effects of directory reconfiguration on its distributed operation. Mechanisms were specified to support the management of distribution knowledge and replication agreements. The conclusions to this work are presented in section 8.3 below.

A further goal of this work has been the implementation of a prototype Directory service. Prototyping has formed an integral part of the design process and has tested many of the ideas generated during specification work. In addition, it has demonstrated the use of existing software tools to support Directory services. The conclusions of prototyping are presented in section 8.4.

8.2. The management of directory information

The Directory stores information reflecting the state of real world communication entities. The issue of information management concerns the update of this information so that it continues to accurately describe these entities as they change over time.

The update of information is achieved by applying operations to add, delete and modify directory entries. Successful management requires that these updates are applied sensibly so that they leave information in an accurate and meaningful state. This requires additional support from directory management mechanisms.

This thesis has specified two management mechanisms belonging to the directory global conceptual model.

- The *data access control* mechanism protects against the *illegal* update of information by constraining who can apply which updates to specified information.
- The *data integrity* mechanism protects against the *invalid* update of information by ensuring that updates adhere to pre-defined integrity rules governing the structure and contents of information.

Before discussing these mechanisms in greater detail, the following paragraphs describe the concepts of *Information Management Domains* and *dynamic management*, critical to their design.

Information Management Domains

Chapter 4 noted that directory information will be updated according to many different management policies corresponding to different administrations. This leads to the definition of the *Information Management Domain* (IMD) as a logical space defining a specific *management framework* of update policies.

Although IMDs are not explicitly represented within the directory model, directory information management tools must be flexible enough to support many external IMDs defining a wide variety of management policies. Chapter 4 noted that IMDs are likely to reflect organisational hierarchy which, in turn, is described by the structure of the Directory Information Tree.

Dynamic management

The Directory will support many services, not all of which will be standardised. The introduction of non standard services requires the definition of new management policies. Furthermore, changes to organisational structure and policy require changes to existing information management policies. These factors imply that the Directory must allow the *dynamic* definition of management policies to reflect changes to the external world. In particular, the current X.500 approach of defining information structures and other management policies via the standardisation process is not acceptable.

Support for IMDs and dynamic management policies have been major requirements of the directory data access control and data integrity mechanisms. The following sections summarise these mechanisms and emphasise how this support is provided.

8.2.1. The data access control mechanism

The data access control mechanism governs which directory users may perform which actions on what information. Section 4.2 discussed several approaches to access control and proceeded to specify a mechanism based on the use of *Access Control Lists* (ACLs). However, other approaches such as capability based mechanisms might also be suitable. Whichever mechanism is chosen must be flexible enough to support the varying access policies of different IMDs and to allow the dynamic management of the access controls themselves.

The ACL based mechanism described in chapter 4 meets these requirements in the following ways.

- ACLs describe *access groups* of users in terms of *Object Set Descriptors* (OSDs). Each OSD names a subtree of the DIT and further defines the access group via a *filter*.

The use of DIT subtrees reflects organisational structure and hence the structures of IMDs.

- ACLs may be created, removed and replaced via the *Add Entry* and *Modify Entry* operations. This allows the dynamic update of access policies to reflect changes in IMD policies.

In addition, the access control mechanism supports flexible *public* or *anonymous* access to information.

8.2.2. The data integrity mechanism

This thesis specified a data integrity mechanism implementing integrity constraints within the Directory service. The integrity mechanism introduces two new structures, specifying the structure of information and constraining its values.

- *Attribute definitions* define the structure and contents of attributes.
- *Entry definitions* define the structure of entries and specify possible directory name forms.

The following conclusions can be drawn about the nature and use of entry and attribute definitions.

Each attribute definition defines a single attribute class and labels it with a human readable *type*. It also declares the definition within a named subtree of the DIT called its *scope*. The scope defines the subtree in which attributes of this class may exist and therefore specifies the locus of effect of the attribute definition. In addition, the attribute definition declares the syntax of attributes of this class and may place limits on the contents of attribute values.

Each entry definition defines a single object class within the directory. It assigns a *type* and *scope* with similar meanings to those in attribute definitions. The contents of entries are constrained in terms of *optional* and *mandatory* attributes and name forms are specified by a combination of possible *superior* object classes and *relative distinguished name* attributes. Furthermore, entry definitions may be nested using the *contains* constructor.

The importance of scope

The use of scope is important in allowing dynamic management and reflecting the policies of different Information Management Domains. Support for these requirements is discussed in the following paragraphs.

The dynamic definition of management policies is supported by several new directory operations allowing the creation, deletion and suspension of entry and attribute definitions via the Directory Access Protocol. The implementation of these operations is possible due to the

presence of scope information defining their logical area of effect. For example, scope constrains the portion of the DIT which needs to be checked before a definition can be deleted.

Scope is also instrumental in reflecting the management policies of different IMDs. IMDs may often be represented by subtrees of the DIT and hence can define their own information structures by creating entry and attribute definitions with local scope. For example, an organisation can define integrity constraints controlling local information in a specific DIT subtree.

Human readable attribute types

Another important aspect of dynamic management is the inclusion of human readable attribute "types" within the directory information model as opposed to within the external models of applications. The inclusion of meaningful types within attribute definitions allows user interfaces to deal with unknown attribute classes by retrieving type and structure information from the Directory system. This is contrary to the view of X.500 where the mapping to human readable attribute types occurs within each user interface.

The inclusion of human readable attribute types supports user interfaces in the handling of previously unseen attributes. This is important for human use of the Directory where the user's external model is not bound to any particular application or service.

8.2.3. Final conclusions for information management

In summary, this thesis draws the following specific conclusions concerning information management within the Directory service.

- The directory information model should include data integrity and access control mechanisms.
- These mechanisms should allow the dynamic definition of access and integrity constraints in order to reflect changing real world management policies.
- These mechanisms must reflect the fact that directory information will be administered according to the policies of many different Information Management Domains.
- Integrity rules may be represented by attribute definitions and entry definitions.
- The ability to define entry and attribute definitions within a specified scope is important in reflecting IMDs and allowing dynamic management.
- The inclusion of human readable attribute types within the directory model supports the dynamic definition of new attribute classes.

- Describing access groups in terms of subtrees of the DIT reflects the administrative hierarchy present in many IMDs.

These conclusions are essential to a Directory service supporting the management of information within an environment of expanding applications, organisations and users.

8.3. The operation and management of the Directory system

Chapters 5 and 6 of this thesis concerned the distributed aspects of the Directory service. Chapter 5 developed a distributed directory model and explored the realisation of directory functionality by a set of autonomous DSAs. Chapter 6 extended this model to support the replication of information between DSAs.

This work has also addressed issues concerning the management of the Directory system. Management of the Directory system describes the problems of maintaining distribution knowledge and replication agreements as the configuration of DSAs alters. These problems belong to the local conceptual layer of the directory architecture and are separate to the information management issues belonging to the global conceptual layer.

The following sections present the conclusions arising from this work.

- Section 8.3.1 reviews the distributed directory model and discusses *knowledge management*.
- Section 8.3.2 describes the distributed execution of operations.
- Section 8.3.3 describes the use and management of replication.

8.3.1. Knowledge and its management

The Directory Information Tree is partitioned between a set of DSAs, each of which is responsible for a number of *fragments*. The responsibilities of a DSA are described by its *reference* containing its name, address and a list of the fragments it maintains.

Navigation is the process of locating the DSA responsible for a specified target entry and occurs as a sequence of *navigation steps* implemented by separate DSAs in a *navigation sequence*. The Directory supports distributed navigation, meaning that each DSA is able to perform a navigation step independently of other DSAs. It does this by using its local *knowledge* of the configuration of the Directory system.

Knowledge is the information, present at each DSA, describing the responsibilities of other DSAs in terms of a set of *references*. Knowledge is therefore the glue binding together the Directory system. This thesis distinguishes three types of knowledge, acquired and managed

in distinct ways.

- A DSA's *minimal* knowledge ensures the fundamental connectivity between DSAs. It consists of the set of references to all *adjacent* DSAs and therefore guarantees that navigation can always occur.
- A DSA's *opportune* knowledge is used to optimise the navigation process and increase the robustness of directory operation. It consists of a set of references to non-adjacent DSAs and is acquired by ad-hoc methods such as caching.
- A DSA's *replicated* knowledge also consists of references to non-adjacent DSAs. However, it is acquired by the establishment of replication agreements between DSAs.

The management of knowledge requires that references are updated to reflect the introduction and removal of DSAs as well as changes to their responsibilities. Without knowledge management, knowledge inconsistencies may arise. For example, a DSA may reference non-existent DSAs or associate existing DSAs with the wrong fragments. These inconsistencies may result in inefficient navigation or, in pathological cases, navigation loops and the segregation of the Directory system. Each of the three types of knowledge described above is acquired and managed in a different way.

The management of minimal knowledge

Minimal knowledge is loaded into a DSA when it is initialised and guarantees that navigation can occur. It is therefore essential that minimal knowledge is maintained in a consistent state at all times. This requires that each change to a DSA's status or responsibilities is propagated to all other DSAs referencing it within their minimal knowledge. This may be achieved by means of a knowledge management protocol, based on the *Update Reference* operation described in section 5.3.2. A strict protocol for minimal knowledge management is possible because a reconfigured DSA can always determine which other DSAs to update from its own minimal knowledge.

The management of opportune knowledge

Opportune knowledge is typically acquired by a number of ad-hoc methods such as caching and is generally used to optimise the navigation process. It is virtually impossible for a DSA to determine which other DSAs hold opportune references to it and, consequently, the management of opportune knowledge cannot be supported by a well defined protocol as was the case with minimal knowledge. Instead, opportune knowledge is treated as unreliable and navigation must be prepared for inconsistent opportune references. To this end, chapter 5 specified the *hop count*, *intended reference* and *route checking* mechanisms to detect

navigation errors. These mechanisms utilise a minimal amount of navigation information attached to operations and represent efficient and reliable methods of error detection. Once an error is detected, navigation continues using minimal knowledge alone. Furthermore, the erroneous opportune reference may often be corrected or deleted automatically without the need for human intervention.

The management of replicated knowledge

Replicated knowledge is acquired during the replication of information between DSAs. Replicated knowledge may be subject to a number of update policies as described in section 8.3.3 below and may therefore exhibit various degrees of reliability.

8.3.2. The execution of distributed operations

Chapter 5 also investigated the execution of operations by the Directory system including the distributed support required for access and integrity controls. A major goal of this work was to minimise the number of DSA interactions required to execute each operation therefore increasing the efficiency of the Directory system.

Section 5.4 specified *operation envelopes* and *result envelopes* containing the information required for the detection of navigation errors. In addition, this information serves to spread opportune knowledge throughout the Directory system.

Distributed operations were described in terms of a *navigation phase*, locating a responsible DSA, and an *execution phase* where the responsible DSA coordinates other DSAs in executing the operation. The execution phases of different operations were examined and the following conclusions were reached.

- The checking of subtree membership within Access Control Lists can be performed local to the DSA containing the relevant ACL. However, the matching of filters requires interaction with the DSA responsible for the user's entry. Navigation to this DSA reduces to one step providing its reference is included within the operation envelope.

- The update of information requires frequent reference to entry and attribute definitions. The efficiency of distributed updates can therefore be greatly improved by copying definitions to all DSAs responsible for entries within their scope. This downward replication of definitions ensures that any DSA implementing an update has local copies of the relevant entry and attribute definitions and does not need to interact with a remote DSA to obtain them. This is supported by the concept of *scope* as described above.

The downward replication of entry and attribute definitions imposes a large update overhead on the system during their creation and deletion. However, this is expected to occur relatively infrequently. A number of new Directory Service Protocol operations were proposed to handle this replication and their distributed execution was also examined.

This work also included a discussion of the *chaining*, *DUA referral* and *DSA referral* modes of directory operation concluding that all three should be supported and used under different circumstances. However, use of the DUA referral mode is generally undesirable because it does not support the automatic propagation and correction of knowledge as described above.

8.3.3. Replication

Chapter 6 discussed the replication of information between DSAs and proposed a general directory replication model. This model describes an abstract replication strategy, capable of supporting many possible implementations.

The Directory system should support replication for the following two reasons.

- Replication enhances performance by copying information from one DSA to another thus reducing the number of DSA interactions required during the execution of operations.
- Replication improves the robustness of the Directory by increasing the availability of information.

The major problem to be addressed is that of consistently updating all copies of replicated information. Previous research in this area has resulted in two broad approaches to this problem.

- The first is the *single master update* mechanism where every update is initially directed at a single master DSA and then propagated to all slave DSAs. A variant of this is the *snapshot* mechanism where updates are slave and not master initiated. The single master approach is simple and suitable for a *loosely coupled* system. However, it does not increase the availability of information for update and therefore does not increase the robustness of the system as a whole. Furthermore, single master mechanisms exhibit *transient inconsistency* where different copies of information may be temporarily inconsistent.
- The second approach is that of *multiple master update* where updates may be directed at more than one copy of information. This increases the robustness of the system but introduces a large update overhead. This thesis has reviewed two multiple master update mechanisms based on *locking* and *timestamping* respectively. Both of these mechanisms eliminate the problem of transient inconsistency.

The proposed directory replication model adopts a hybrid approach, utilising a combination of single master and multiple master mechanisms to increase performance and robustness respectively. The model divides the Directory system into clusters of DSAs exhibiting the following characteristics.

- The DSAs within a cluster are multiple masters for the same fragments and are tightly coupled by a local multiple master update mechanism.
- Replication between clusters is supported by a single master update mechanism where the master DSA is, in fact, a master cluster.

Replication within a cluster increases the robustness of the system. It is expected that a cluster will contain only a few DSAs connected by fast and reliable communication links, although the future introduction of high speed WAN might relax this constraint. The choice of multiple master update mechanism is local to each cluster.

Replication between clusters increases the performance of the Directory system. Section 6.5 described the support required for a single master mechanism. This includes the following features.

- Inter-cluster replication is controlled by *replication agreements* between DSAs describing the information to be replicated and whether updates are to be master or slave initiated.
- The information to be replicated is described in terms of *Object Set Descriptors*. These are logical statements of replication policy allowing the master DSA to automatically determine which information should be replicated as the DIT changes.
- Replicated information may be restricted to *naming information only* in order to increase the efficiency of navigation with a minimal update overhead. This defines the *replicated knowledge* mentioned previously. Furthermore, replicated knowledge is maintained via the single master update mechanism.

The automatic management of replicated information is an important feature of the above work.

New distributed operations were proposed to support the establishment and management of replication agreements as well as the propagation of updates. Chapter 6 also discussed the need for crash recovery mechanisms within the Directory system.

8.3.4. Final conclusions for system operation and management

In summary, the following conclusions can be drawn from chapters 5 and 6 which examined the distributed operation of the Directory service.

- The distributed navigation of operations requires the presence of *knowledge* describing the responsibilities of DSAs.
- Knowledge can be divided into the categories of *minimal*, *opportune* and *replicated* where each category is acquired and managed differently.
- Knowledge management mechanisms are required to support the continued operation of the Directory system following its reconfiguration.
- Minimal knowledge is managed by a well defined protocol.
- Opportune knowledge is considered unreliable and is managed by ad-hoc error detection and correction mechanisms.
- Replicated knowledge is managed by the general directory replication mechanism.
- Distributed operations should be wrapped in *envelopes* containing the information required for detecting navigation errors. This information also facilitates the automatic propagation of opportune knowledge.
- The efficiency of distributed access control is increased by including the reference of the user's home DSA within operation envelopes.
- The efficiency of distributed integrity control is increased by the downward replication of entry and attribute definitions to all DSAs within their scopes.
- Replication increases the efficiency and robustness of distributed operation.
- The Directory should support a hybrid replication model using both single and multiple master update mechanisms.
- Single master replication is described by replication agreements where the information to be copied is specified by Object Set Descriptors.
- Multiple master replication occurs within small clusters of DSAs. The choice of multiple master update mechanism is specific to each cluster.

8.4. The implementation of a prototype Directory service

The implementation of a prototype Directory service has formed an integral part of this research and was undertaken for the following three reasons.

1. Testing and refining the results of specification work.
2. Exploring the use of the *RTI Ingres* and *ISO Development Environment* (ISODE) software tools for supporting Directory services.

3. Indicating issues to be addressed by production Directory services.

Implementation work was subject to both time and resource constraints and the functionality of the prototype was therefore limited to a subset of that specified within this thesis.

A Directory System Agent was implemented using the RTI Ingres relational database management system as a storage mechanism. This DSA supports the general retrieval and modification of information as well as access control and integrity mechanisms. It also provides a distributed *Read Entry* operation including navigation and knowledge management mechanisms.

Several Directory User Agents were implemented supporting both human and application use of the Directory. Part of this work occurred within the European Community COST-11-TER funded *AMIGO MHS+* project where the prototype was used to support a pilot distribution list protocol.

The full functionality and design of the prototype Directory service were described in chapter 7. This work resulted in the following conclusions.

- Both the data access control and integrity mechanisms specified within chapter 4 are implementable and are necessary to support information management even on a small scale.
- The use of the prototype to support the AMIGO MHS+ distribution list protocol demonstrated the importance of allowing the dynamic definition of entry and attribute definitions.
- ISODE was vital to the construction of the prototype and similar tools will play a major role in the future development of distributed applications.
- Ingres provided a flexible prototyping tool and facilitated the rapid implementation of the Directory System Agent. However, relational database management systems might not be suitable for production directories, due to performance problems.
- The use of caching and replication techniques will be necessary to improve the performance of production systems.
- The design of sophisticated user interfaces will play a major role in supporting human use of the Directory.

The previous sections have presented the conclusions of this research. The remainder of this chapter describes the implications of these conclusions for X.500 and outlines unresolved issues and possible future directions for Directory services in general.

8.5. Implications of this research for X.500

This thesis has used various X.500 directory models as a base for exploring a number of management issues, not addressed by the standard.

This research occurred in parallel with the development of 1988 X.500 and attempted to keep track of the standard as it changed. The result is that many of the conclusions of the previous sections can be applied directly to X.500. This section therefore describes the broad implications which this work could have for the future development of the X.500 standard. This is particularly relevant at the present time as the start of the 1988-92 study period approaches.

Before going into further detail, the following section briefly outlines the major differences in approach between X.500 and this thesis.

8.5.1. The approach of X.500

The ISO and CCITT standardisation bodies initiated separate studies into Directory services in 1984. However, 1986 saw them combining their efforts to produce a joint standard. At this time, the model reached its greatest complexity and the 1986 Melbourne ISO/CCITT output included an access control model and even basic support for replication [CCITT-XDS86]. Due to the pressing timescale, this model was drastically simplified at the following meetings and many of the more complex issues were designated *for further study*. Work from that point to the present day has aimed at solidifying this simpler model although, perhaps surprisingly, it has since been extended by the introduction of *schemas*.

The current status of X.500 therefore appears to be that of a stop gap standard with the intention of satisfying the urgent requirement for some form of global Directory service. Many issues have been deliberately left until the next study period.

Another factor influencing the design of X.500 is that it has been developed by standards bodies. The result of this obvious fact is that X.500 appears oriented towards supporting other standardised services, perhaps at the cost of lack of support for non-standard services and local applications. In particular, it has been shaped by the requirements of X.400, the best established of the communication standards.

Conversely, the work in this thesis has emphasised the use of Directory services by non-standard applications, such as the AMIGO distribution list protocol and has viewed the Directory in the context of general distributed databases. This difference in approach has resulted in conclusions which might provide useful input for the next ISO/CCITT study period. These conclusions are presented below in the form of six extensions to 1988 X.500.

8.5.2. Extensions to X.500

The X.500 standard was reviewed in section 1.4 of this thesis. It defines entries, attributes, the Directory Information Tree and operations to read, search and modify information. In addition, it specifies aspects of navigation and distributed operations as well as defining a number of standard attribute types and object classes.

Extension 1: Dynamic schemas

The only support for information management within X.500 is provided by *schemas*, similar in purpose to the entry and attribute definitions defined in chapter 4. However, X.500 does not allow the dynamic creation and deletion of schemas. Instead, a number of standard attribute types and object classes are specified in the X.520 (*The Directory - Selected Attribute Types*) and X.521 (*The Directory - Selected Object Classes*) parts of the standard. Application specific types may also be defined within standards for other services. This allows new applications to specify their own schemas, although these are not generally visible to the outside world.

There are no abstract operations supporting dynamic schemas and no concept of *scope* to make such operations implementable.

The current X.500 approach may be sufficient for standardised applications defining their own schemas on paper. However, it will not adequately support many non-standard applications emerging in the future. This thesis therefore recommends that the dynamic management of schemas is introduced into X.500 in the form of new abstract operations belonging to a management port.

Extension 2: Scope for schemas

The lack of a *scope* for attribute types and object classes in X.500 means that there is no support for different Information Management Domains. Once again, this does not encourage non-standard or local use of the Directory. In order to allow the definition of local management policies, X.500 should therefore support the concept of scope.

This thesis proposes that X.500 introduces scoping for schemas. In general, it may be sufficient to introduce scope as an optional property of schema definitions.

Extension 3: Human readable attribute types within the information model

X.500 identifies attribute types and object classes by means of *Object Identifiers* (OIDs) which are globally unique hierarchical structures forming their own OID tree. However, OIDs are not meaningful to humans and a mapping to human readable types has to occur at

the user interface to the Directory. Yet again, this is acceptable for standardised applications dealing with a constant set of pre-defined attributes. It is not suitable for human use of the Directory where new attribute types are defined dynamically. In this case, the user interface needs a clue as to how to present new attributes to human users. This can be achieved by allowing suggested human labels for attributes to be included in their definitions. This thesis suggests that this mechanism is included in X.500 with the proviso that these labels are optional and can, of course, be ignored by the interface.

Extension 4: Support for knowledge management

Part X.518 of X.500 (*The Directory - Procedures for Distributed Operation*) defines its own knowledge and navigation model. It mentions the need for *knowledge administration* and suggests that this may be achieved by two methods.

- Firstly, a DSA may update its reference at other DSAs whenever it changes in some way.
- Secondly, DSAs may request each other's *cross references* thus refreshing their own knowledge.

X.500 also mentions that knowledge inconsistencies should be detected and, if possible, corrected. However, the precise methods for doing this are left as "local matters" and are said to be "outside the scope" of the standard.

The problem is that distributed navigation may involve many different DSAs under different administrations. It will be difficult to detect and correct errors occurring between DSAs from different administrations using purely local methods.

This thesis therefore suggests that X.500 should include standard support for knowledge error detection and correction. This support could be used by all DSAs to implement robust error detection and correction mechanisms. In particular, X.500 might support the *hop-count, intended reference* and *route checking* mechanisms of chapter 5.

Extension 5: access controls

1988 X.500 will not support a standardised access control mechanism. Instead, access control is left as a local matter and Annex F to part X.501 (*The Directory - Models*) outlines possible approaches to the problem.

Global access control requires a global access control framework supported by all DSAs. In particular, the realistic implementation of distributed access control will require some standardised support within the Directory system. A good example is the need for user information in operation envelopes to make filter checking feasible (section 5.4.1 of this thesis).

This thesis therefore notes that a standardised directory access control framework should be considered within the next study period. The distributed support required for access control should also be examined.

Extension 6: Replication

Chapter 6 demonstrated the need for the Directory service to support the replication of information between DSAs. 1988 X.500 will not include a replication mechanism and this issue should therefore be addressed within the next study period. A model based on DSA clusters as specified within chapter 6 could provide a useful basis for this work.

This concludes the discussion of the relationship between this thesis and the future development of X.500. The following sections turn their attention to the future of Directory services. Firstly, there will be a discussion of immediate issues to be addressed by research in this area. Secondly, the future role of the Directory service will be explored, particularly its role as a Management service.

8.6. Unresolved issues

This section notes a number of issues remaining unresolved by this thesis. These issues present immediate problems to the large scale implementation of Directory services and therefore should be the subject of short term research.

8.6.1. Dynamic attribute syntax

One task of attribute definitions is to specify the syntax of attribute values. Section 4.3.3 specified a limited choice of possible syntaxes for this purpose. However, there is clearly a requirement for more application specific syntaxes. For example, a telephone number can be defined as a structured sequence of digits and a X.400 Originator/Recipient address is structured as an ordered sequence of X.400 specific attributes. This requirement for new application specific syntaxes will grow as new applications are developed.

This thesis has proposed that the Directory service should support the dynamic definition of attribute and entry definitions. Surely it should also allow the dynamic definition of new attribute syntaxes in order to support new applications. Perhaps new syntaxes could be specified using the ASN.1 notation thus allowing arbitrarily complex structures to be defined. As a trivial example, consider defining an employee number as an ordered sequence of two alphabetic characters followed by six digits and then an arbitrary character

string.

The problems of dynamically defining, storing and interpreting arbitrarily complex attribute syntaxes are many. In effect, dynamic syntax definition allows users to redefine aspects of the Directory Access Protocol during the operation of the Directory. This may require sophisticated functionality in DUAs and DSAs to interpret the protocol (i.e. a dynamically programmable ASN.1 parser). These issues deserve consideration in the near future.

8.6.2. Third party access control

Chapter 4 described an access control mechanism controlling the actions of both human and application users of the Directory. In fact, the general approach of this thesis has been that the Directory treats applications and humans identically from the access point of view.

Section 1.1 suggested that future communication would be provided by the cooperation of many services. Perhaps the treatment of applications should be reconsidered in the light of this observation. The *third party* access control problem, described below, is one symptom of general problems in this area.

Consider the simple example of a Message Handling System (MHS) interacting with the Directory service to expand a message sent to a distribution list by a human user. One can envisage two access control scenarios.

- In the first case, the MTA expanding the message interacts with the Directory as itself, irrespective of who the initial user is. The Directory therefore controls access based on the identity of the MTA. This implies that the user might obtain information via the MHS that they may not have obtained by direct access to the Directory. It also suggests that the MHS must support its own access control mechanism.
- In the second case, the MTA acts on behalf of the initial user and the Directory controls access according to their identity. This implies that the MTA must be capable of assuming the user's identity. Furthermore, it suggests that the ability to use the MHS depends upon a user's access rights within the Directory service.

For this example, the first scenario seems intuitively correct. However, the general principles involved are not clear, particularly where many different services are involved in complex interactions.

This *third party* problem clearly belongs in a wider context than just that of the Directory service. In fact, these issues truly belong to areas such as *Distributed Office Architectures* (DOA) [ISO-DOA88]. However, research is required to determine the effect of these broader issues on directory access control. For example, allowing applications to impersonate humans might suggest the use of a capability based access control mechanism within

the Directory service.

8.6.3. Directory system access control

The access control mechanism described in chapter 4 belongs to the directory global conceptual layer. It controls access to the abstract directory information base and is not concerned with lower layer distribution issues. Particularly important is the concept of *location transparency*, implying that a user has the same access rights from all locations within the distributed system.

The Directory service is implemented by a set of DUAs and cooperating DSAs provided by different, perhaps even competing, organisations. This thesis has assumed that the DSAs constituting the Directory system trust each other implicitly once mutual authentication has occurred. In reality, this is not likely to be the case.

In general, there may be a need for some form of system level access control mechanism describing the information which DSAs are prepared to share and the degree to which they will cooperate. The possible effects of such a mechanism are indicated by the following two examples.

- Suppose that a DSA requests a replication agreement with another DSA, provided by a competing company. It is unlikely that the agreement would be permitted if it involved the replication of sensitive information. How can this policy be implemented by the system? Replication is an area clearly requiring system level access control.
- Imagine the case where a user has permission to access information stored within their home DSA. What happens if, when visiting a competing company, they attempt to chain a query to their home DSA via a foreign DSA? Even though authentication has occurred, would the home DSA be prepared to return the requested information via the foreign DSA? Would the user see different results from different access points?

These examples suggest that, in many cases, DSAs may not trust each other and may not be willing to share information. This implies the need for some form of access control mechanism at the directory local conceptual layer even if it remains local to each DSA. Furthermore, the second example shows that access to the Directory service might not be locationally transparent.

System level access control and its implications for users have not been discussed within this thesis. However, they clearly require study if commercial companies are to become involved in a global Directory service.

8.7. Future directions for Directory services

This section examines the future role of Directory services and outlines a number of issues for longer term research.

- Section 8.7.1 examines the relationship between the Directory service and the ISO's emerging OSI *management framework*.
- Section 8.7.2 examines the extension of distributed naming, beyond communication entities, to general *objects* inhabiting distributed systems.

8.7.1. The relationship of the Directory to OSI management

Network and systems management is a broad topic and is the subject of current research within several groups [JEF88, ZIM88, BAC88]. In particular, the ISO have recently begun the development of a general OSI *management framework* defining a number of *Specific Management Functional Areas* (SMFA) for future research. [ISO-OSI-4]. Although still in a primative state, this framework and its SMFA aim to provide a model of OSI network and systems management covering the following issues:

- Fault management
- Accounting management
- Configuration and name management
- Performance management
- Security management

This thesis has discussed a number of management issues specific to the Directory service including both name and configuration management.

There is clearly a relationship between the work in this thesis and the goals of the OSI management framework. In fact, there is a general inter-dependency between Directory services and Management services [WEIS88].

- Firstly, the Directory is itself a Management service. More specifically, Management will utilise the Directory for the storage of certain management information.
- Secondly, like other services, the Directory will utilise Management services for its operation and maintenance.

These issues are explored below.

Use of the Directory by Management Services

The Directory service manages names and communication information and is likely to be the first available Management service. There are many similarities between Directory and OSI management issues. Specifically, the management framework defines a *Management Information Base* (MIB) of information "relating to managed objects".

The structure of the MIB has yet to be determined. However, its specification may cover many issues already addressed within Directory services. From the point of view of this thesis, issues such as the dynamic definition of information structures have strong parallels with the Management Information Base and the structure of management information. Consequently, those groups studying management issues should pay close attention to the results emerging from Directory research.

Although it may play a major role within management, the Directory cannot be the only Management service. In particular, it exhibits the following limitations:

- It is unsuitable for storing highly dynamic information. For example, network performance, accounting and fault statistics.
- It is unable to represent complex relationships between entities. For example, it cannot naturally represent the many-to-many relationship between end-users and hosts on a Local Area Network.

These limitations imply that future research must develop additional Management services able to handle these types of information.

Use of Management services by the Directory

Chapters 5 and 6 of this thesis discussed a number of issues concerning Directory system management. These issues clearly fall within the broader context of general systems management and should be covered by the OSI management framework. Due to the current lack of a management framework, this thesis has proposed a number of independent solutions for Directory management. However, its future arrival will require re-evaluation of this work.

In addition, this thesis provides a specific example of the management of a distributed system. This might form useful input for management groups.

In summary, there is a strong mutual dependency between the Directory service and emerging work on network and systems management. This suggests several directions for future research in this area:

- The development of a *management framework* using concepts from Directory research.
- The development of additional Management services to handle those aspects of management where the Directory is limited.
- Managing the Directory using the future management framework.

8.7.2. Naming distributed objects

Current Directory research is primarily concerned with the distributed naming of *communication entities* such as users, application entities and hosts. However, these entities represent a fraction of the *objects* existing within many distributed environments. In general, distributed systems manipulate a wide range of objects such as files, messages and documents. Like communication entities, these objects require naming.

The distributed naming and management of objects is becoming a critical issue within several areas of research including *computer mediated communication* and *object oriented distributed systems*. The following paragraphs examine the naming requirements of these environments and demonstrate that the current approach to Directory services is not sufficient to meet their needs.

Computer mediated communication

Recent studies have produced several models for distributed *computer mediated communication*. These can be divided into two broad categories:

- Procedural models concerning the structured flow of messages between communication partners playing *roles* within a communication structure or activity [BOW88, DAN88].
- Information based approaches, considering the shared access of structured information by groups of users [BOG88].

The procedural models specify distributed *objects* such as *roles*, *rules*, *activities* and *tasks*. However, these models have not yet addressed the distributed naming of these objects. This will clearly become a critical issue.

The AMIGO MHS+ information based approach includes a *data model* specifying abstract information structures representing objects such as *messages*, *documents*, *conferences* and *conversations* [BENF88b]. This work briefly considers the naming of these objects. However, no firm conclusion is reached.

The objects specified by these models differ from communication entities in several crucial respects.

- They may be highly dynamic in nature. For example, the contents of a conference may change rapidly.
- They may be identified by different names within different environments. For example a document may have one name within the AMIGO project and another when viewed from the ISO environment.
- There are complex relationships between the objects. For example, the mapping between users, roles, activities and rules.

The naming and management of these objects requires a Directory service supporting the following functionality:

- The management of dynamic objects.
- Context or environment based naming.
- The ability to represent, search and manipulate the complex relationships between objects.

This functionality is not possible using current directory models due to the loosely coupled nature of the Directory and the use of hierarchical, as opposed to graphical naming. Furthermore, the issue of distributed naming has not been addressed in detail by those groups working within this area. Consequently, future research must develop new naming schemes and Directories suitable for these environments.

Object oriented distributed systems

Recent years have seen the application of *object oriented* models to several areas of computer science such as programming languages, databases and distributed systems. In particular, recent research has considered the design of general distributed systems based on object oriented frameworks [HORN88, MUL87, BAN88]. These systems may play a major role in the future development of distributed computing and communications.

The introduction of distributed object oriented models will have a large impact on distributed naming and, hence, Directory services. For example, a recent paper by Stefani describes a general model for distributed object oriented systems, including the following features. [STEF88].

All entities within the distributed system are represented as *objects*. Thus, objects could be relational tuples, documents, fragments of code or data servers. The advantage of describing all entities as objects is that common mechanisms may be applied to their operation and

interaction. Thus, the first point arising from this work is that there might be a very wide range of objects to be named within future distributed systems.

Objects generally contain both data and operations acting on that data. Thus, although some objects may be passive, many objects will be *active*, invoking operations on other objects. Consequently, the contents of objects might be highly dynamic.

The objects within a distributed system are divided into generic classes. In general, classes may contain nested sub-classes with inheritance of properties between them. Thus, there may be complex relationships between objects.

A major issue to be addressed within distributed object oriented systems is the management of objects. This includes their naming. Stefani describes objects as belonging to distributed *management units* responsible for their naming and location. An important feature of this work is that each management unit might support its own naming context and name resolution function.

The conclusion of the above points is that the naming of distributed objects will become a critical issue as object oriented concepts play an increasing role within future distributed systems. These systems will require the development of Directory services, able to meet the following requirements:

- The naming of a wide variety of objects.
- The management of highly dynamic objects.
- The representation of complex relationships between objects.
- Support for many different naming contexts and mechanisms.

These requirements clearly fall beyond the capabilities of present directory technology. Consequently, future research into Directory services should consider the naming of objects within general object oriented distributed systems and, in particular, should address the above issues.

In summary, the communication entities considered by this thesis represent a fraction of the objects requiring naming within a distributed environment. Unlike communication entities, the objects belonging to future group communication and object oriented systems might be highly dynamic and complex in structure. Current directory models are not suitable for naming such objects. Thus, future research should develop naming schemes and Directories meeting the requirements of these environments. This might require the extension of current models or the design of entirely new services.

8.8. Final word

Successful communication is vital to the achievement of human tasks and goals. The advent of computer networks has opened the door on a new era where electronic media and services improve the quality of human communication, allowing it to occur in ways not previously possible. Use of current services, such as electronic mail, has already demonstrated the potential of computer based communication. These systems will seem primative compared to the applications of the future.

Communication requires information. Furthermore, this information requires careful management if successful communication is to occur on a global scale. The need for information management has become critical as the number of users and services has increased and the introduction of a global Directory service is now vital to the continued growth of electronic communication.

This thesis has described mechanisms allowing the Directory to support information and system management and therefore meet the above need. Hopefully, the near future will see the introduction of these ideas into the X.500 standard where they are currently unsupported.

Beyond this, the Directory should be viewed as a first step towards the establishment of a general *management framework*. The long term future should see the introduction of additional management services to meet requirements beyond those of the Directory. These services can then be integrated into the management framework, providing an environment in which computer based communication can continue to grow.

Appendix A

Specification of Directory Protocols

Chapters 3 through 6 of this thesis specified many abstract operations belonging to the *Directory Access Protocol* (DAP) and *Directory Service Protocol* (DSP). This appendix draws this work together and presents a detailed specification of these operations using the ISO *Abstract Service Definition Conventions* [ISO-ASD88].

Functionality is defined in terms of a set of *abstract ports* supporting sets of *abstract operations*. It is important to note that these ports and operations are abstract structures and do not describe a specific Directory implementation.

Some of the ports described below use the same names as those within the X.500 Directory abstract service definition [CCITT-X511]. In general, these ports have similar meanings to those within X.500. However, their functionality is not identical and they should not be confused with their X.500 counterparts.

Figure 9.1 provides an overview of the Directory service and system in terms of the abstract ports supported by DUAs and DSAs. The operations provided by these ports are described by the subsequent paragraphs.

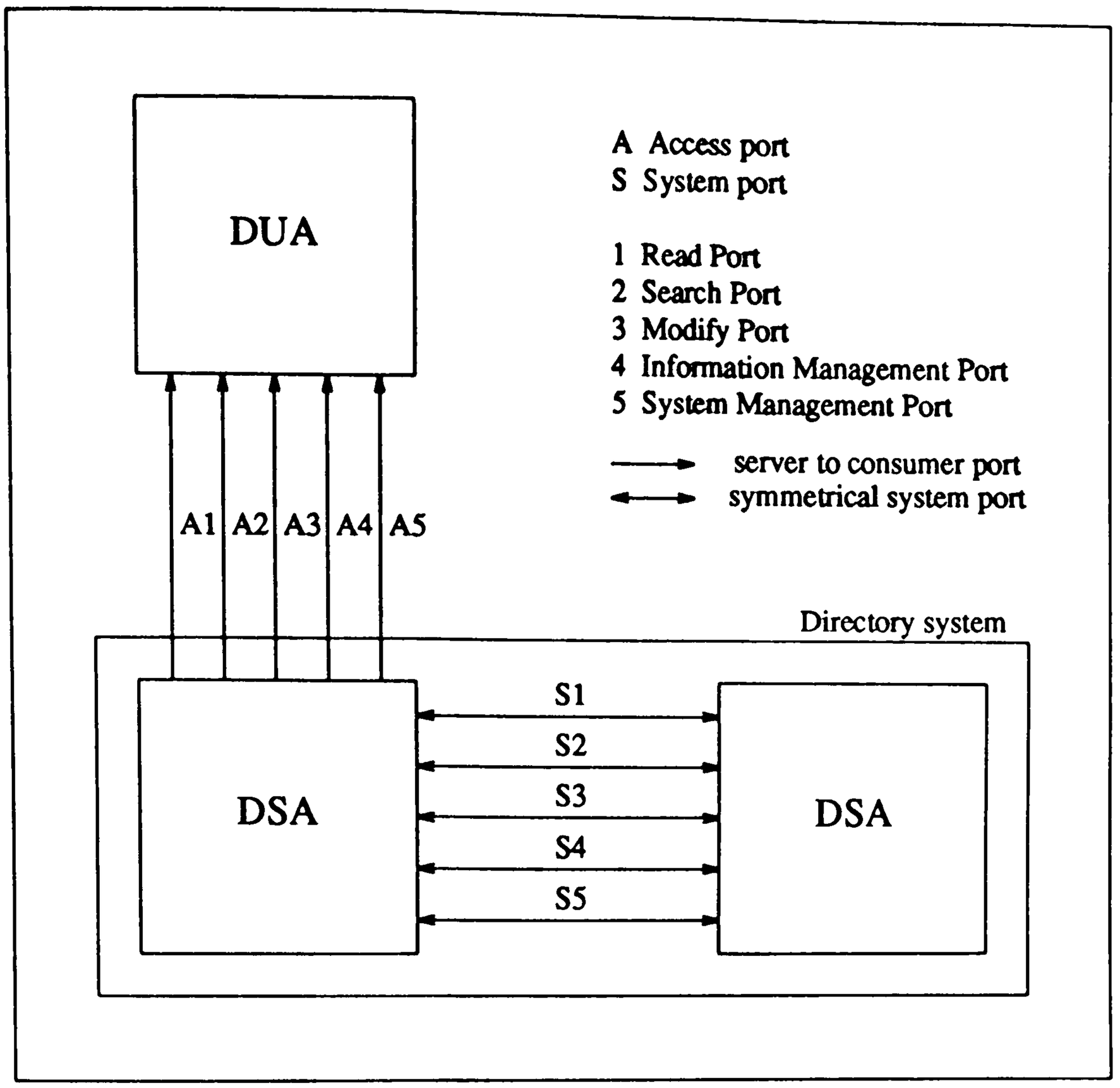


Figure 9.1: The Directory system and service

DUA Abstract Service Definition

The DUA Abstract Service Definition defines the Directory access protocol in terms of the access ports and operations listed below. For each of these ports, the DUA acts as the service consumer and the DSA as the service provider.

The most notable aspects of this definition are the inclusion of the *Information Management* and *System Management* ports.

- The Information Management port supports those operations responsible for the dynamic manipluation of entry and attribute definitions.
- The System Management port supports those operations allowing administrators to manage replication agreements.

These operations were specified by chapters 3 and 4 of this thesis.

DUA Abstract Service Definition	
port	operation
Read (A1)	Read Entry
Search (A2)	List Subordinates Search
Modify (A3)	Modify Entry Add Entry Delete Entry Add Alias Delete Alias Suspend Entry Reinstate Entry Reset Alias
Information Management (A4)	Add Attribute Def Delete Attribute Def Suspend Attribute Def Reinstate Attribute Def Read Attribute Def Add Entry Def Delete Entry Def Suspend Entry Def Reinstate Entry Def Read Entry Def
System Management (A5)	Establish Replication Terminate Replication

DSA Abstract Service Definition

In addition to acting as the service provider for the above ports, the DSA supports the Directory Service Protocol in terms of the following system ports and operations. These ports are *symmetic* meaning that each DSA may act as both service provider and consumer.

The following table indicates that the DSP supports distributed versions of the DAP operations. In addition, other operations provide the following functionality:

- Checking filters against remote entries.
- Checking whether entry and attribute definitions are "in use" and propagating their addition and deletion between DSAs.
- Updating and deleting references.
- Updating replicated information.

These operations were specified by chapters 5 and 6 of this thesis.

DSA Abstract Service Definition	
port	operation
Distributed Read (S1)	Distributed Read Entry Check Filter Retrieve Reference
Distributed Search (S2)	Distributed List Subordinates Search
Distributed Modify (S3)	Distributed versions of operations from A3 e.g. Distributed Add Entry Distributed Modify Entry ...
Distributed Information Management (S4)	Check Definition Use New Entry Def New Attribute Def Delete Definition Suspend Definition Reinstate Definition Distributed versions of operations from A4 e.g. Distributed Add Attribute Def ...
Distributed System Management (S5)	Distributed Establish Replication Distributed Terminate Replication Update replication Refresh Replication Update Reference Delete Reference

ASN.1 description of abstract ports and operations

The remainder of this appendix presents an ASN.1 definition of the ports and operations described above. It is important to note that ASN.1 descriptions were used throughout this thesis to clarify the meaning of data structures and operations and not to define a protocol ready for implementation. Consequently, the following ASN.1 summary of this work does not define a complete and detailed protocol.

```
— DUA ABSTRACT SERVICE DEFINITION

— DUA OBJECT

dua OBJECT
        PORTS {
                Read [C],
                Search [C],
                Modify [C],
                InformationManagement [C],
                SystemManagement [C] }
        ::= id-ob-dua

— DUA PORTS

Read PORT
        CONSUMER INVOKES {
                ReadEntry }
        ::= id-pt-read

Search PORT
        CONSUMER INVOKES {
                ListSubordinates, Search }
        ::= id-pt-search

Modify PORT
        CONSUMER INVOKES {
                ModifyEntry, AddEntry, DeleteEntry, AddAlias, DeleteAlias,
                SuspendEntry ReinstateEntry, ResetAlias }
        ::= id-pt-modify

InformationManagement PORT
        CONSUMER INVOKES {
                AddAttributeDef, DeleteAttributeDef, SuspendAttributeDef,
                ReinstateAttributeDef, ReadAttributeDef, AddEntryDef,
                DeleteEntryDef, SuspendEntryDef, ReinstateEntryDef,
                ReadEntryDef }
        ::= id-pt-inf

SystemManagement PORT
        CONSUMER INVOKES {
                EstablishReplication, TerminateReplication }
        ::= id-pt-sys


— ATTRIBUTES AND NAMES

AttributeType ::= PrintableString

AttributeValue ::= CHOICE {
```

```
                integer [0] INTEGER,
                real [1] REAL,
                characterstring [2] PrintableString,
                name [3] Name,
                bitstring [4] BITSTRING,
                octetstring [5] OctetString
                boolean [6] BOOLEAN,
                time [7] GeneralisedTime }

NamingAttribute ::= SEQUENCE {
                        type AttributeType,
                        value PrintableString }

DistinguishedName ::= SEQUENCE OF NamingAttribute

Name ::= SEQUENCE OF NamingAttribute

-- ACCESS CONTROLS, OBJECT SET DESCRIPTORS AND FILTERS

AccessControlList ::= SET OF AccessControlListElement

AccessControlListElement ::= SEQUENCE {
                        categories BITSTRING { detect(0), read(1), etc. },
                        accessgroup CHOICE {
                            public [0] NULL,
                            users [1] ObjectSetDescriptor } }

ObjectSetDescriptor ::= SET {
                        subtree [0] DistinguishedName,
                        level [1] CHOICE {
                            fixed [0] Integer,
                            unlimited [1] NULL DEFAULT },
                        filter [2] Filter OPTIONAL }

Filter ::= CHOICE {
                        item [0] SEQUENCE {
                            AttributeType,
                            AttributeValue },
                        and [1] SET OF Filter,
                        or [2] SET OF Filter,
                        not [3] Filter }

-- ENTRY AND ATTRIBUTE DEFINITIONS

AttributeDefinition ::= SET {
                        type [0] PrintableString,
                        scope [1] DistinguishedName,
                        recurring [2] BOOLEAN,
```

```
                        structure [3] AttributeStructure }

AttributeStructure ::= CHOICE {
                        integer [0] IntegerConstraint,
                        real [1] RealConstraint,
                        characterstring [2] CharConstraint,
                        name [3] NameConstraint,
                        bitstring [4] NULL,
                        octetstring [5] NULL,
                        boolean [6] NULL,
                        time [7] NULL }

IntegerConstraint ::= SEQUENCE {
                        minimum INTEGER OPTIONAL,
                        maximum INTEGER OPTIONAL,
                        range SET OF INTEGER }


RealConstraint ::= SEQUENCE {
                        minimum REAL OPTIONAL,
                        maximum REAL OPTIONAL,
                        range SET OF REAL }

CharConstraint ::= SET OF PrintableString

NameConstraint ::= SET OF DistinguishedName

EntryDefinition ::= SET {
                        type [0] PrintableString,
                        scope [1] DistinguishedName,
                        contains [2] ObjectClass OPTIONAL,
                        superiors [3] SET OF ObjectClass,
                        rdntypes [4] SET OF AttributeTemplate,
                        mandatory [5] SET OF AttributeTemplate OPTIONAL,
                        optional [6] SET OF AttributeTemplate OPTIONAL,
                        defaultacl [7] AccessControlList OPTIONAL,
                        acl [8] AccessControlList OPTIONAL }


ObjectClass ::= SEQUENCE {
                        type PrintableString,
                        scope DistinguishedName }

AttributeTemplate ::= SEQUENCE {
                        type PrintableString,
                        scope DistinguishedName,
                        defaultvalue AttributeValue OPTIONAL,
                        defaultacl AccessControlList OPTIONAL }
```

```
— OPERATIONS

— READ ENTRY: read attributes from a named entry

ReadEntry ::= ABSTRACT–OPERATION
                    ARGUMENT ReadArgument
                    RESULT ReadResult
                    ERRORS { NameError, AttributeError, AccessError }

ReadArgument ::= SET {
                        entry [0] Name,
                        types [1] SET OF AttributeType,
                        info–requested [2] BITSTRING { atts (0), acls (1), defs (2) } }

ReadResult ::= SET {
                        [0] EntryInformation OPTIONAL,
                        [1] SET OF ReadError }

EntryInformation ::= SET {
                        name [0] DistinguishedName,
                        class [1] ObjectClass,
                        entrylevelacl [2] AccessControlList OPTIONAL,
                        attributes [3] SET OF AttributeInformation }

AttributeInformation ::= SET {
                        type [0] AttributeType,
                        definition [1] AttributeDefinition OPTIONAL,
                        attributelevelacl [2] AccessControlList OPTIONAL,
                        values [3] SET OF AttributeValue }

ReadError ::= CHOICE {
                        [0] NameError,
                        [1] AttributeError,
                        [2] AccessError }


— LIST SUBORDINATES: list children of a named parent

ListSubordinates ::= ABSTRACT–OPERATION
                    ARGUMENT ListArgument
                    RESULT ListResult
                    ERRORS { NameError, AccessError }

ListArgument ::= Name

ListResult ::= SET OF DistinguishedName
```

```
— SEARCH: read from entries matching a filter in subtrees

Search ::= ABSTRACT–OPERATION
                ARGUMENT SearchArgument
                RESULT SearchResult
                ERRORS { NameError, AttributeError, AccessError }

SearchArgument ::= SET {
                        subtree [0] Name,
                        level [1] CHOICE {
                                fixed [0] INTEGER,
                                unlimited [1] NULL DEFAULT },
                        filter [2] Filter,
                        targets [3] SET OF AttributeType,
                        info–requested [4] BITSTRING { atts (0), acls (1), defs (2) } }

SearchResult ::= SET {
                        entries [0] SET OF EntryInformation,
                        errors [1] SET OF ReadError }


— MODIFY ENTRY: add, replace, delete attributes within named entry

ModifyEntry ::= ABSTRACT–OPERATION
                ARGUMENT ModifyArgument
                RESULT ModifyResult
                ERRORS { NameError, AttributeError,
                                ModifyError, AccessError }

ModifyArgument ::= SET {
                        entry [0] Name,
                        entry–acl [1] AccessModification OPTIONAL,
                        modifications [2] SET OF Modification }

Modification ::= SEQUENCE {
                        type AttributeType,
                        attribute–acl AccessModification OPTIONAL,
                        CHOICE {
                                add [0] AttributeValue,
                                delete [1] AttributeValue,
                                replace [2] SEQUENCE {
                                        oldvalue AttributeValue,
                                        newvalue AttributeValue } } }

AccessModification ::= CHOICE {
                        acl [0] AccessControlList,
                        empty [1] NULL }

ModifyResult ::= NULL
```

```
— ADD ENTRY: add a new entry to the DIT

AddEntry ::= ABSTRACT–OPERATION
                    ARGUMENT AddEntryArgument
                    RESULT AddEntryResult
                    ERRORS { NameError, RDNError,
                                        AttributeError, AccessError }

AddEntryArgument ::= SET {
                                parent [0] Name,
                                rdn [1] SET OF NamingAttribute,
                                entry–acl [2] AccessControlList OPTIONAL,
                                initial–info [3] SET OF InitialAttribute }

InitialAttribute ::= SEQUENCE {
                                Attribute,
                                AccessControlList OPTIONAL }

AddEntryResult ::= NULL


— DELETE ENTRY: delete an entry from the DIT

DeleteEntry ::=  ABSTRACT–OPERATION
                    ARGUMENT DeleteEntryArgument
                    RESULT DeleteEntryResult
                    ERRORS { NameError, AccessError }

DeleteEntryArgument ::= Name

DeleteEntryResult ::= NULL


— ADD ALIAS: add a new alias to the DIT

AddAlias ::= ABSTRACT–OPERATION
                    ARGUMENT AddAliasArgument
                    RESULT AddAliasResult
                    ERRORS { NameError, RDNError, AccessError }

AddAliasArgument ::= SET {
                                parent [0] Name,
                                rdn [1] SET OF NamingAttribute,
                                aliased–object [2] Name }

AddAliasResult ::= NULL
```

```
-- DELETE ALIAS: delete an alias from the DIT

DeleteAlias ::=  ABSTRACT-OPERATION
                 ARGUMENT DeleteAliasArgument
                 RESULT DeleteAliasResult
                 ERRORS { NameError, AccessError }

DeleteAliasArgument ::= Name

DeleteAliasResult ::= NULL

-- SUSPEND ENTRY: allows an entry to be suspended

SuspendEntry ::=  ABSTRACT-OPERATION
                  ARGUMENT SuspendEntryArgument
                  RESULT SuspendEntryResult
                  ERRORS { NameError, AccessError, SuspendError }

SuspendEntryArgument ::= SET {
                         entry [0] Name,
                         new-name [1] DistinguishedName,
                         date [2] GeneralisedTime }

SuspendEntryResult ::= NULL

-- REINSTATE ENTRY: resinstates a suspended entry

ReinstateEntry ::=  ABSTRACT-OPERATION
                    ARGUMENT ReinstateEntryArgument
                    RESULT ReinstateEntryResult
                    ERRORS { NameError, AccessError, SuspendError }

ReinstateEntryArgument ::= Name

ReinstateEntryResult ::= NULL

-- RESET ALIAS: restes an alias to point at a new entry

ResetAlias ::=  ABSTRACT-OPERATION
                ARGUMENT ResetAliasArgument
                RESULT ResetAliasResult
                ERRORS { NameError, AccessError, AliasError }

ResetAliasArgument ::= SET {
                       alias [0] Name,
                       new-object [1] DistinguishedName }
```

```
ResetAliasResult ::= NULL


— ATTRIBUTE DEFINITION OPERATIONS

— READ ATTRIBUTE DEF: read attribute definitions

ReadAttributeDef ::=  ABSTRACT–OPERATION
                      ARGUMENT ReadAttributeDefArgument
                      RESULT ReadAttributeDefResult
                      ERRORS { NameError, AccessError, AttributeDefError }

ReadAttributeDefArgument ::= SET {
                             type [0] PrintableString OPTIONAL,
                             scope [1] Name }

ReadAttributeDefResult ::= SET OF AttributeDefinition


— ADD ATTRIBUTE DEF: add a new attribute definition

AddAttributeDef ::=  ABSTRACT–OPERATION
                     ARGUMENT AddAttributeDefArgument
                     RESULT AddAttributeDefResult
                     ERRORS { NameError, AccessError, AttributeDefError }

AddAttributeDefArgument ::=  AttributeDefinition

AddAttributeDefResult ::= NULL


— DELETE ATTRIBUTE DEF: delete an attribute definition

DeleteAttributeDef ::=  ABSTRACT–OPERATION
                        ARGUMENT DeleteAttributeDefArgument
                        RESULT DeleteAttributeDefResult
                        ERRORS { NameError, AccessError, AttributeDefError }

DeleteAttributeDefArgument ::= SET {
                               type [0] PrintableString,
                               scope [1] Name }

DeleteAttributeDefResult ::= NULL


— SUSPEND ATTRIBUTE DEF: suspend an attribute definition

SuspendAttributeDef ::=  ABSTRACT–OPERATION
                         ARGUMENT SuspendAttributeDefArgument
                         RESULT SuspendAttributeDefResult
```

```
                    ERRORS { NameError, AccessError, AttributeDefError }

SuspendAttributeDefArgument ::= SET {
                        type [0] PrintableString,
                        scope [1] Name }

SuspendAttributeDefResult ::= NULL


-- REINSTATE ATTRIBUTE DEF: reinstate an attribute definition

ReinstateAttributeDef ::=  ABSTRACT-OPERATION
                    ARGUMENT ReinstateAttributeDefArgument
                    RESULT ReinstateAttributeDefResult
                    ERRORS { NameError, AccessError, AttributeDefError }

ReinstateAttributeDefArgument ::= SET {
                        type [0] PrintableString,
                        scope [1] Name }

ReinstateAttributeDefResult ::= NULL


-- ENTRY DEFINITION OPERATIONS

-- READ ENTRY DEF: read entry definitions

ReadEntryDef ::=  ABSTRACT-OPERATION
                    ARGUMENT ReadEntryDefArgument
                    RESULT ReadEntryDefResult
                    ERRORS { NameError, AccessError, EntryDefError }

ReadEntryDefArgument ::= SET {
                        type [0] PrintableString OPTIONAL,
                        scope [1] Name }

ReadEntryDefResult ::= SET OF AttributeDefinition


-- ADD ENTRY DEF: add a new entry definition

AddEntryDef ::=  ABSTRACT-OPERATION
                    ARGUMENT AddEntryDefArgument
                    RESULT AddEntryDefResult
                    ERRORS { NameError, AccessError, EntryDefError }

AddEntryDefArgument ::= EntryDefinition

AddEntryDefResult ::= NULL
```

```
— DELETE ENTRY DEF: delete an entry definition

DeleteEntryDef ::=  ABSTRACT–OPERATION
                    ARGUMENT DeleteEntryDefArgument
                    RESULT DeleteEntryDefResult
                    ERRORS { NameError, AccessError, EntryDefError }

DeleteEntryDefArgument ::= SET {
                               type [0] PrintableString,
                               scope [1] Name }

DeleteEntryDefResult ::= NULL


— SUSPEND ENTRY DEF: suspend an entry definition

SuspendEntryDef ::=  ABSTRACT–OPERATION
                     ARGUMENT SuspendEntryDefArgument
                     RESULT SuspendEntryDefResult
                     ERRORS { NameError, AccessError, EntryDefError }

SuspendEntryDefArgument ::= SET {
                                type [0] PrintableString,
                                scope [1] Name }

SuspendEntryDefResult ::= NULL


— REINSTATE ENTRY DEF: reinstate an entry definition

ReinstateEntryDef ::=  ABSTRACT–OPERATION
                       ARGUMENT ReinstateEntryDefArgument
                       RESULT ReinstateEntryDefResult
                       ERRORS { NameError, AccessError, EntryDefError }

ReinstateEntryDefArgument ::= SET {
                                  type [0] PrintableString,
                                  scope [1] Name }

ReinstateEntryDefResult ::= NULL


— REPLICATION MANAGEMENT OPERATIONS

— ESTABLISH REPLICATION: initialise a replication agreement

EstablishReplication ::=  ABSTRACT–OPERATION
                          ARGUMENT EstablishReplicationArgument
                          RESULT EstablishReplicationResult
                          ERRORS { NameError, AccessError, ReplicationError }
```

```
EstablishReplicationArgument ::= SET {
                                    policy [0] SET OF ObjectSetDescriptor,
                                    information [1] INTEGER { full (0), naming (1) },
                                    update–style [2] INTEGER { master (0), slave (1) } }

EstablishReplicationResult ::= INTEGER


— TERMINATE REPLICATION: terminate a replication agreement

TerminateReplication ::=  ABSTRACT–OPERATION
                          ARGUMENT TerminateReplicationArgument
                          RESULT TerminateReplicationResult
                          ERRORS { NameError, AccessError, ReplicationError }

TerminateReplicationArgument ::= INTEGER

TerminateReplicationResult ::= NULL


— ERRORS

NameError ::= ABSTRACT–ERROR
                  PARAMETER SET {
                                    problem [0] INTEGER { no–match (0) } }

AttributeError ::= ABSTRACT–ERROR
                  PARAMETER SET {
                                    problem [0] INTEGER { attribute–missing (0) },
                                    type [1] AttributeType }

AccessError ::= ABSTRACT–ERROR
                  PARAMETER SET {
                                    problem [0] INTEGER { rights–insufficient (0) }

ModifyError ::= ABSTRACT–ERROR
                  PARAMETER SET {
                                    problem [0] INTEGER { no–change (0), non–leaf (1),
                                                          already–exists (2), is–alias (3),
                                                          illegal–change (4) },
                                    type [1] AttributeType,
                                    value [2] AttributeValue }

RDNError ::= ABSTRACT–ERROR
                  PARAMETER SET {
                                    problem [0] INTEGER { invalid–type (0) },
                                    type [1] AttributeType }

SuspendError ::= ABSTRACT–ERROR
                  PARAMETER SET {
```

```
                        problem [0] INTEGER { already-suspended (0), not-suspended (1),
                                              date-problem (2) } }

AliasError ::= ABSTRACT-ERROR
        PARAMETER SET {
                problem [0] INTEGER { not-alias (0) }

AttributeDefError ::= ABSTRACT-ERROR
        PARAMETER SET {
                problem [0] INTEGER { non-unique (0), invalid-scope (1),
                                      still-active (2), violation (3) }

EntryDefError ::= ABSTRACT-ERROR
        PARAMETER SET {
                problem [0] INTEGER { non-unique (0), invalid-scope (1),
                                      still-active (2), type-mandatory (3),
                                      type-undefined (4), superior-invalid (5),
                                      definition-invalid (6) },
                objectclass [1] ObjectClass,
                type [2] AttributeType }

ReplicationError ::= ABSTRACT-ERROR
        PARAMETER SET {
                problem [0] INTEGER { refused (0), invalid-policy (1) }

-- DSA ABSTRACT SERVICE DEFINITION

-- DSA OBJECT

dsa OBJECT
        PORTS {
                Read [S] VISIBLE,
                Search [S] VISIBLE,
                Modify [S] VISIBLE,
                InformationManagement [S] VISIBLE,
                SystemManagement [S] VISIBLE,

                DistributedRead PAIRED WITH dsa,
                DistributedSearch PAIRED WITH dsa,
                DistributedModify PAIRED WITH dsa,
                DistributedInformationManagement PAIRED WITH dsa,
                DistributedSystemManagement PAIRED WITH dsa }
::= id-ob-dsa
```

```
— DSA DISTRIBUTED PORTS
— includes distributed versions of DUA operations

DistributedRead PORT
        ABSTRACT–OPERATIONS {
                DistributedReadEntry, CheckFilter, RetrieveReference }
        ::= id–pt–disread

DistributedSearch PORT
        ABSTRACT–OPERATIONS {
                DistributedSearch, etc. }
        ::= id–pt–dissearch

DistributedModify PORT
        ABSTRACT–OPERATIONS {
                DistributedModifyEntry, etc. }
        ::= id–pt–dismodify

DistributedInformationManagement PORT
        ABSTRACT–OPERATIONS {
                DistributedAddAttributeDef, etc.
                CheckDefinitionUse, NewEntryDef, NewAttributeDef,
                DeleteDefinition, SuspendDefinition, ReinstateDefinition }
        ::= id–pt–disinf

DistributedSystemManagement PORT
        ABSTRACT–OPERATIONS {
                DistributedEstablishReplication, etc.
                UpdateReplication, RefreshReplication,
                UpdateReference, DeleteReference }
        ::= id–pt–dissys


— DATA STRUCTURES

— REFERENCES

Reference ::= SEQUENCE {
                dsa DistinguishedName,
                address — application entity address,
                responsibilities SET OF DistinguishedName }

— OPERATION AND RESULT ENVELOPES

UserInformation ::= CHOICE {
                public [0] NULL,
                identified [1] SEQUENCE {
                        user DistinguishedName,
                        users–dsa Reference } }
```

```
OperationEnvelope ::= SET {
                        minimal-only [0] BOOLEAN DEFAULT FALSE,
                        hop-count [1] INTEGER,
                        intended-dsa [2] Reference,
                        last-dsa [3] Reference,
                        sending-dsa [4] Reference,
                        aliases-seen [5] SET OF DistinguishedName,
                        user-information [6] UserInformation }

ResultEnvelope ::= SET {
                        result-type [0] BITSTRING,
                        sending-dsa [1] Reference,
                        intended-dsa [2] Reference }

-- REPLICATION UPDATES

Update ::= CHOICE {
                        add-entry [0] EntryInformation,
                        add-alias [1] AliasInformation,
                        delete-entry [2] DistinguishedName,
                        add-continuation [3] ContinuationPoint,
                        delete-continuation [4] DistinguishedName,
                        modify-attributes [5] Modification }

ContinuationPoint ::= SEQUENCE {
                        DistinguishedName,
                        Reference OPTIONAL }

AliasInformation ::= SEQUENCE {
                        alias DistinguishedName,
                        object DistinguishedName }


-- OPERATIONS

-- CHECK FILTER: matches a filter against a remote user's entry

CheckFilter ::= ABSTRACT-OPERATION
                    ARGUMENT CheckFilterArgument
                    RESULT CheckFilterResult
                    ERRORS { DSAError, NavigationError, Referral }

CheckFilterArgument ::= SET {
                        user [0] DistinguishedName,
                        filter [1] Filter,
                        envelope [2] OperationEnvelope }

CheckFilterResult ::= SET {
                        match-result [0] BOOLEAN,
                        envelope [1] ResultEnvelope }
```

```
— RETRIEVE REFERENCE: finds the reference of a user's home DSA

RetrieveReference ::= ABSTRACT–OPERATION
                      ARGUMENT RetrieveReferenceArgument
                      RESULT RetrieveReferenceResult
                      ERRORS { DSAError, NavigationError, Referral }

RetrieveReferenceArgument ::= SET {
                                  user [0] DistinguishedName,
                                  envelope [1] OperationEnvelope }

RetrieveReferenceResult ::= SET {
                                  home–dsa [0] Reference,
                                  envelope [1] ResultEnvelope }


— CHECK DEFINITION USE: check whether a definition is in use in a subtree

CheckDefinitionUse ::= ABSTRACT–OPERATION
                       ARGUMENT CheckDefinitionUseArgument
                       RESULT CheckDefinitionUseResult
                       ERRORS { DSAError, NavigationError, Referral }

CheckDefinitionUseArgument ::= SET {
                                  type [0] PrintableString,
                                  scope [1] Name,
                                  def–type [2] INTEGER { attribute (0), entry (1) },
                                  envelope [3] OperationEnvelope }

CheckDefinitionUseResult ::= SET {
                                  result [0] BOOLEAN,
                                  envelope [1] ResultEnvelope }


— NEW ATTRIBUTE DEF: propagate an attribute def downwards

NewAttributeDef ::= ABSTRACT–OPERATION
                    ARGUMENT NewAttributeDefArgument
                    RESULT NewAttributeDefResult
                    ERRORS { DSAError, NavigationError, Referral }

NewAttributeDefArgument ::= SET {
                                definition [0] AttributeDefinition,
                                envelope [1] OperationEnvelope }

NewAttributeDefResult ::= ResultEnvelope
```

```
-- NEW ENTRY DEF: propagate an entry def downwards

NewEntryDef ::= ABSTRACT-OPERATION
                ARGUMENT NewEntryDefArgument
                RESULT NewEntryDefResult
                ERRORS { DSAError, NavigationError, Referral }

NewEntryDefArgument ::= SET {
                        definition [0] EntryDefinition,
                        envelope [1] OperationEnvelope }

NewEntryDefResult ::= ResultEnvelope


-- DELETE DEFINITION: propagate the deletion of a definition

DeleteDefinition ::= ABSTRACT-OPERATION
                     ARGUMENT DeleteDefinitionArgument
                     RESULT DeleteDefinitionResult
                     ERRORS { DSAError, NavigationError, Referral }

DeleteDefinitionArgument ::= SET {
                             type [0] PrintableString,
                             scope [1] Name,
                             def-type [2] INTEGER { attribute (0), entry (1) },
                             envelope [3] OperationEnvelope } }

DeleteDefinitionResult ::= ResultEnvelope


-- SUSPEND DEFINITION: propagate the suspension of a definition

SuspendDefinition ::= ABSTRACT-OPERATION
                      ARGUMENT SuspendDefinitionArgument
                      RESULT SuspendDefinitionResult
                      ERRORS { DSAError, NavigationError, Referral }

SuspendDefinitionArgument ::= SET {
                              type [0] PrintableString,
                              scope [1] Name,
                              def-type [2] INTEGER { attribute (0), entry (1) },
                              envelope [3] OperationEnvelope } }

SuspendDefinitionResult ::= ResultEnvelope


-- REINSTATE DEFINITION: propagate a reinstated definition

ReinstateDefinition ::= ABSTRACT-OPERATION
                        ARGUMENT ReinstateDefinitionArgument
```

```
                    RESULT ReinstateDefinitionResult
                    ERRORS { DSAError, NavigationError, Referral }

ReinstateDefinitionArgument ::= SET {
                        type [0] PrintableString,
                        scope [1] Name,
                        def–type [2] INTEGER { attribute (0), entry (1) },
                        envelope [3] OperationEnvelope } }

ReinstateDefinitionResult ::= ResultEnvelope


— UPDATE REPLICATION: master initiated update of replicated information

UpdateReplication ::= ABSTRACT–OPERATION
                    ARGUMENT UpdateReplicationArgument
                    RESULT UpdateReplicationResult
                    ERRORS { DSAError, NavigationError, Referral }

UpdateReplicationArgument ::= SET {
                        agreement–id [0] INTEGER,
                        updates [1] SET OF Update,
                        envelope [2] OperationEnvelope } }

UpdateReplicationResult ::= ResultEnvelope


— REFRESH REPLICATION: slave initiated update of replicated information

RefreshReplication ::= ABSTRACT–OPERATION
                    ARGUMENT RefreshReplicationArgument
                    RESULT RefreshReplicationResult
                    ERRORS { DSAError, NavigationError, Referral }

RefreshReplicationArgument ::= SET {
                        agreement–id [0] INTEGER,
                        envelope [2] OperationEnvelope } }

RefreshReplicationResult ::= SET {
                        updates [0] SET OF Update,
                        envelope [1] ResultEnvelope }


— UPDATE REFERENCE: update a DSA's reference at a remote DSA

UpdateReference ::=  ABSTRACT–OPERATION
                    ARGUMENT UpdateReferenceArgument
                    RESULT UpdateReferenceResult
                    ERRORS { DSAError, ReferenceError, Referral }
```

```
UpdateReferenceArgument ::= SET {
                        old [0] Reference,
                        new [1] Reference,
                        envelope [2] OperationEnvelope }

UpdateReferenceResult ::= SET {
                        reference [0] Reference,
                        envelope [1] ResultEnvelope }
```

— DELETE REFERENCE: delete a DSA's reference at a remote DSA

```
DeleteReference ::=  ABSTRACT–OPERATION
                ARGUMENT DeleteReferenceArgument
                RESULT DeleteReferenceResult
                ERRORS { DSAError, ReferenceError, NavigationError, Referral }

DeleteReferenceArgument ::= SET {
                        reference [0] Reference,
                        envelope [1] OperationEnvelope }

DeleteReferenceResult ::= ResultEnvelope
```

— DISTIBUTED VERSIONS OF DAP OPERATIONS

In general, each DAP operation is mapped into a DSP counterpart by the addition of operation and result envelopes. An example *Read Entry* to *Distributed Read Entry* mapping is shown below.

```
DistributedReadEntry ::=  ABSTRACT–OPERATION
                ARGUMENT DistributedReadArgument
                RESULT DistributedReadResult
                ERRORS { DSAError, ReferenceError, NavigationError, Referral,
                                NameError, AccessError, AttributeError }

DistributedReadArgument ::= SET {
                        entry [0] Name,
                        types [1] SET OF AttributeType,
                        envelope [3] OperationEnvelope }

DistributedReadResult ::= SET {
                        [0] EntryInformation OPTIONAL,
                        [1] SET OF ReadError,
                        [2] ResultEnvelope }
```

— *ERRORS*

```
DSAError ::= ABSTRACT-ERROR
             PARAMETER SET {
                          problem [0] INTEGER { refused (0) } }

ReferenceError ::= ABSTRACT-ERROR
             PARAMETER SET {
                          problem [0] INTEGER { incorrect-reference (0) },
                          reference [1] Reference }

NavigationError ::= ABSTRACT-ERROR
             PARAMETER SET {
                          problem [0] INTEGER { fatal-error (0), non-fatal-error (1) }
                          reference [1] Reference OPTIONAL }

Referral ::= ABSTRACT-ERROR
             PARAMETER SET {
                          reference [0] Reference }
```

Bibliography

ADI80.

M. E. Adiba and B.G. Lindsay, "Database Snapshots," *Proceedings of the 6th International Conference on Very Large Databases*, October 1980.

ANSI75.

ANSI/X3/SPARC Study Group on Database Management Systems: Interim Report, American National Standards Institute (ANSI), 1975.

BAC88.

J.M. Bacon, C Horn, A. Langsford, S.J. Mullender and W. Zimmer, "MANDIS - Architectural Basis for Management," *Research into Networks and Distributed Systems (Proceedings of EUTECO 88)*, North Holland, 1988.

BAN88.

J.P. Banatre, M. Banatre and G. Muller, "Main Aspects of the GOTHIC Distributed System," *Research into Networks and Distributed Applications (proceedings of EUTECO 88)*, North Holland, 1988.

BENF86a.

S. Benford and A. Warnking, "The Meaningful use of Object Set Descriptors within Access Control Lists," *Thorn Document: DFN-17*, GMD, St. Augustin, Bonn, West Germany, November 1986. (also contributed to the 1986 ISO/CCITT Egham meeting)

BENF86b.

S. Benford and A. Warnking, "A Default Access Rule for when Access Rights are Not Specified," *Thorn document: DFN-8*, GMD, St. Augustin, Bonn, West Germany, August 1986. (also contributed to the 1986 ISO/CCITT Egham meeting)

BENF87.

Steve Benford and Julian Onions, "Pilot Distribution Lists - Agents and Directories," *Proceedings of the IFIP 6.5 International Working Conference on MHS*, North Holland, April 1987.

BENF88a.

Steve Benford, Julian Onions, Manfred Bogen and Bernd Wagner, "The Implementation of AMIGO Distribution Lists," *Research into Networks and Distributed Applications (proceedings of EUTECO 88)*, North Holland, April 1988.

3ENF88b.

S. Benford, "A Data Model for the AMIGO Communications Environment," *Research Into Networks and Distributed Applications (proceedings of EUTECO 88)*, North Holland, April 1988.

BENF88c.

Steve Benford, "Dynamic Definition of Entries and Attributes in the Directory Service," *Proceedings of the 8th International Conference on Distributed Computing Systems*, IEEE Computer Society Press, San Jose, CA, USA, June 1988.

BENF88d.

Steve Benford, "Navigation and Knowledge Management within a Distributed Directory System," *Proceedings of the IFIP 6.5 Working Conference on Message Handling Systems*, North Holland, October 1988. (to be published)

BERK86.

UNIX User's Reference Manual, Computer Science Division, University of California, USA, Berkeley CA, USA, April 1986.

BERN81.

P.A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," *Computing Surveys*, vol. 13, no. 2, pp. 185-221, June 1981.

BERN82.

P.A. Bernstein and N. Goodman, "A Sophisticate's Introduction to Distributed Database Concurrency Control," *Proceedings of the 8th VLDB Conference*, pp. 62-76, September 1982.

BIR81.

Andrew D. Birrell, Roy Levin, Roger M Needham and Michael D. Schroeder, *Grapevine*, Xerox Palo Alto Research Center, Palo Alto, USA, April 1981.

BOG88.

Manfred Bogen and Karl-Heinz Weiss, "Group Co-ordination in a Distributed Environment," *Research into Networks and Distributed Applications (proceedings of EUTECO 88)*, North Holland, April 1988.

BOW88.

J. Bowers, J. Churcher and T. Roberts, "Structuring Computer-Mediated Communication in COSMOS," *Research Into Networks and Distributed Applications (proceedings of EUTECO 88)*, North Holland, April 1988.

BRY88.

Bill Bryant, *Designing an Authentication System: a Dialogue in Four Scenes*, Project

Athena, MIT, Cambridge, MA, USA, February 1988.

:CITT-ROS86.

Remote Operations: Model, Notation and Service Definition, International Telephone and Telegraph Consultative Committee (CCITT), Geneva, October 1986.

:CITT-X29.

"Procedures for the Exchange of Control Information and User Data between a Packet Assembly/Dissambly Facility (PAD) and a Packet Mode DTE or another PAD," *Data Communications Networks Services and Facilities, Terminal Equipment and Interfaces, Recommendations X.1 - X.29*, International Telegraph and Telephone Consultative Committee (CCITT), Geneva, 1980.

CCITT-X400.

"Message Handling Systems: System Model-Service Elements," *Recommendation X.400*, International Telegraph and Telephone Consultative Committee (CCITT), Malaga-Torremolinos, October 1984.

CCITT-X500.

"Information Processing Systems - Open Systems Interconnection - The Directory - Overview of Concepts, Models and Service," *ISO 9594-1, CCITT X.500*, ISO and CCITT, Gloucester, 1988.

CCITT-X501.

"Information Processing Systems - Open Systems Interconnection - The Directory - Models," *ISO 9594-2, CCITT X.501*, ISO and CCITT, Gloucester, 1988.

CCITT-X511.

"Information Processing Systems - Open Systems Interconnection - The Directory - Abstract Service Definition," *ISO 9594-3, CCITT X.511*, ISO and CCITT, Gloucester, 1988.

CCITT-X518.

"Information Processing Systems - Open Systems Interconnection - The Directory - Procedures for Distributed Operation," *ISO 9594-4, CCITT X.518*, ISO and CCITT, Gloucester, 1988.

CCITT-X519.

"Information Processing Systems - Open Systems Interconnection - The Directory - Protocol Specifications," *ISO 9594-5, CCITT X.519*, ISO and CCITT, Gloucester, 1988.

CCITT-X520.

"Information Processing Systems - Open Systems Interconnection - The Directory -

Selected Attribute Types,'' *ISO 9594-6, CCITT X.520*, ISO and CCITT, Gloucester, 1988.

CCITT-X521.

''Information Processing Systems - Open Systems Interconnection - The Directory - Selected Object Classes,'' *ISO 9594-7, CCITT X.521*, ISO and CCITT, Gloucester, 1988.

CCITT-XDS86.

ISO/CCITT Directory Convergence Documents (X.ds) #1-#8, ISO and CCITT, Melbourne, April 1986.

CHU82.

W.W. Chu and P. Hurley, ''Query Processing for Distributed Database Systems,'' *IEEE Transactions on Computers*, vol. 31, pp. 835-850, IEEE Computer Press, September 1982.

CODA71.

Data Base Task Group of CODASYL Programming Language Committee (Report), April 1971. Available from ACM, BCS and IAG

CODD70.

E.F. Codd, ''A Relational Model of Data for Large Shared Data Bases,'' *Communications of the ACM*, vol. 77, 1970.

CROC82.

David H. Crocker, ''Standard for the Format of ARPA Internet Text Messages,'' *Request For Comments 822*, Network Information Centre, August 1982.

CSCW86.

''CSCW '86: Conference on Computer Supported Cooperative Work,'' *Conference Proceedings*, Austin, Texas, USA, December 1986.

DAN88.

T. Danielsen and U. Pankoke-Babatz, ''The AMIGO Activity Model,'' *Research Into Networks and Distributed Applications (proceedings of EUTECO 88)*, North Holland, April 1988.

DATE77.

C.J. Date, *An Introduction to Database Systems*, Addison-Wesley, 1977. (second edition)

DATE83.

C.J. Date, *An Introduction to Database Systems (volume II)*, Addison-Wesley, 1983.

)DN-TCP.

"Transmission Control Protocol," *DDN Protocol Handbook, Volume One: Military Standard Protocols*, DDN Network Information Center, September 1981. (also available as ARPA Internet RFC 793)

)DN-TELNET.

Telnet Protocol Specification, DDN Network Information Center, May 1983. (also available as ARPA Internet RFC 854)

)DN-UDP.

User Datagram Protocol, DDN Network Information Center, August 1980. (also available as ARPA Internet RFC 768)

)EEN82.

S.M Deen, "A General Framework for the Architecture of Distributed Database Systems," *Distributed Data Sharing Systems*, North Holland, 1982.

DEEN88.

S.M. Deen, M.C Taylor, P.A. Ingram and K.W. Raynor, "A Distributed Directory Database System for Telecommunications," *The Computer Journal*, vol. 31, no. 2, pp. 175-181, 1988.

DEM78.

Richard A. Demillo, David P. Dobkin, Anita K. Jones and Richard J. Lipton (editors), *Foundations of Secure Computation*, Academic Press, 1978.

DEM82.

John C. Demco and Gerald W. Neufeld, *Annotated References on Naming*, Department of Computer Science, University of British Columbia, Vancouver, Canada, October 1982.

DYER87.

Stephen P. Dyer, *Hesiod Name Service: Application Programmer's Guide*, MIT Project Athena, July 1987.

ECMA-TC32.

TC32-TG5: Document Filing and Retrieval, European Computer Manufacturers Association (ECMA), July 1987.

ECMA-TR32.

TR-32: Directory Access and Service Protocol, European Computer Manufacturers Association (ECMA), 1987.

:ND88.

A Endrizzi, "Research Issues in Packet Switched Networks," *Research into Networks and Distributed Applications (proceedings of EUTECO 88)*, North Holland, April 1988.

:PS77.

R. Epstein, "A Tutorial on Ingres," *Memorandum no. ERL-M77-25*, Electronics Research Laboratory, UCLA, Berkeley, CA, USA, December 1977.

EPS78.

R. Epstein, M.R Stonebraker and E. Wong, "Distributed Query Processing in a Relational Data Base System," *Proc 1978 ACM SIGMOD International Conference on Management of Data*, June 1978. (also inlcuded in "The Ingres Papers", Addison-Wesley, 1986)

FER81.

E.B. Fernandez, R.C. Summers and C. Wood, *Database Security and Integrity*, Addison-Wesley, 1981.

HEV82.

A.R. Hevner, "Methods for Data Retrieval in Distributed Systems," *Proceedings of the Second Symposium on Reliability in Distributed Software and Database Systems*, pp. 1-10, 1982.

HORN88.

Chris Horn, "An Object Oriented Model for Distributed Processing," *Research into Networks and Distributed Applications (proceedings of EUTECO 88)*, North Holland, 1988.

HORT87.

Mark R. Horton, "Standard for Interchange of USENET Messages," *Request For Comments 1036*, Network Information Centre (SRI), December 1987.

IBM81.

SQL/Data System, Terminal User's Reference, IBM Corporation, USA, White Plains, NY, 1981. ref: SH24-5017-1

IFIP-DS83.

"Naming, Addressing and Directory Service for Message Handling Systems," *Working Paper N78*, IFIP Working Group 6.5, February 1983.

ISO-ASD88.

"Information Processing Systems - Text Communication - MOTIS - Abstract Service Definition Conventions," *ISO/IEC Draft International Standard 10021-3*,

International Standards Organisation (ISO), Geneva, March 1988.

ISO-ASN86.

"Information Processing - Open Systems Interconnection: Specification of Abstract Syntax Notation One (ASN.1)," *International Standard 8824*, International Standards Organisation (ISO), May 1986.

ISO-DOA88.

"Information Processing - Text Communication - Distributed Office Applications Model," *Draft Proposal: ISO/DP 10031-1/2*, International Standards Organisation (ISO), 1988.

ISO-JTMP.

ISO/TC97/SC21/N210: Information Processing Systems - OSI - Job Transfer and Manipulation Concepts and Services, International Standards Organisation (ISO), February 1985.

ISO-OSI-3.

"OSI Reference Model - Part 3 - Naming and Addressing," *ISO 7498-3*, International Standards Organisation (ISO).

ISO-OSI-4.

"OSI Reference Model - Part 4 - Management Framework," *ISO DIS 7498-4*, International Standards Organisation (ISO), 1988.

ISO-OSI84.

"Information Processing Systems - Open Systems Interconnection: Basic Reference Model," *International Standard 7498*, International Standards Organisation (ISO), October 1984.

ISO-ROS87.

"Information Processing Systems - Text Communication - MOTIS - Remote Operations Part 1: Model, Notation and Service Definition," *Working document for standard 9072/2*, International Standards Organisation, November 1987.

ISO-TP4.

"Information Processing Systems - Open Systems Interconnection: Transport Protocol Specification," *ISO - 8073*, The International Standards Organisation (ISO), July 1986.

JEF88.

Tony Jeffree, "Current and Future Development of OSI Management Standards," *Research Into Networks and Distributed Applications (proceedings of EUTECO 88)*, North Holland, Vienna, 1988.

OHN78.

S.C. Johnson, *Yacc: Yet Another Compiler-Compiler*, Bell Laboratories, Murray Hill, NJ,USA, July 1978.

ERN78.

Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice Hall International Inc., 1978.

IL87a.

S.E. Kille, *Plan for the THORN Large Scale Pilot Exercise*, Department of Computer Science, University College London, London, July 1987. Reference UCL-36

IL87b.

Steve Kille, "MHS use of Directory Services for Routing," *Proceedings of the IFIP 6.5 Working Conference on Message Handling Systems*, North Holland, April 1987.

KIL87c.

Steve Kille, "Access Control," *Internal Note: 2060*, Department of Computer Science, University College London, February 1987. Submission to the ISO/CCITT meeting, Munich 1987

KIL88a.

S.E. Kille, "PP - A Message Transfer Agent," *Proceedings of the IFIP 6.5 Working Conference 1988*, North Holland, October 1988. (to be published)

KIL88b.

S.E. Kille, C.J. Robbins and A. Turland, *The ISO Development Environment: User's Manual (version 4.0)*, July 1988. Volume 5: QUIPU

KIL88c.

S.E. Kille, "The QUIPU Directory Service," *Proceedings of the IFIP 6.5 Working Conference on Message Handling Systems*, North Holland, October 1988. (invited paper)

KIL88d.

S.E. Kille and C.J Robbins, "Distributed Operations in QUIPU," *Esprit Communications Week*, November 1988.

KING84.

Douglas P. Kingston III, "MMDFII: A Technical Review," *Proceedings of the Summer Usenix Conference and Exhibition*, Salt Lake City, Utah, June 1984.

KNI88.

Graham Knight(1), Henry Krecioch(2), Massimo Antonellini(3), *The Practical Implications of OSI Network Management*, (1) UCL, UK, (2) Modcomp, (3) Olivetti/SW,

1988. Issued as part of Project INCA

KOH81.

W.H. Kohler, "A Survey of Techniques for Synchronisation and Recovery in Decentralised Computer Systems," *Computing Surveys*, vol. 13, no. 2, pp. 149-183, June 1981.

LAM78.

L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System," *CACM*, vol. 21, no. 7, July 1978.

LAN83.

L. Landweber, M Litzkow, D. Neuhengen and M. Solomon, *Architecture of the CSNET Name Server*, University of Wisconsin-Madison, USA, 1983.

LANT85.

Keith A. Lantz, Judy L. Edighoffer and Bruce L. Hitson, "Towards a Universal Directory Service," *Proceedings of the 4th PODC Conference*, 1985.

LAR82a.

J. Larmouth, *JNT Name Registration Scheme: Major Design Variables*, Joint Network Team (JNT), August 1982.

LAR82b.

J. Larmouth, *JNT Name Registration Scheme: The NRS Conceptual Schema*, Joint Network Team (JNT), August 1982.

LAR82c.

J. Larmouth, *JNT Name Registration Scheme: Use of the NRS by Remote Sites*, Joint Network Team (JNT), November 1982.

LAR85.

James A. Larson and Saeed Rahimi (editors), *Tutorial: Distributed Database Management*, IEEE Computer Press, 1985.

LENG87.

Thomas Lenggenhager, Bernhard Plattner, Rolf Stadler and Andreas Zogg, "The ISO/CCITT Directory Service as a Distributed Database," *Proceedings of the IFIP 6.5 Working Conference on Message Handling Systems*, North Holland, April 1987.

MILL87.

S.P. Miller, B.C. Neuman, J.I. Schiller and J.H Saltzer, "Kerebos Authentication and Authorisation System," *Project Athena Technical Plan (part E.2.1)*, Project Athena, MIT, Massachusetts, USA, December 1987.

IOCK83a.

P. Mockapetris, "Domain Names: Concepts and Facilities," *Request For Comments 882*, Network Information Center, November 1983.

IOCK83b.

P. Mockapetris, "Domain Names: Implementation and Specification," *Request For Comments 883*, Network Information Center, November 1983.

MOCK84.

P.V Mockapetris, "The Domain Name System," *Computer Based Message Services (Proceedings of the 1984 IFIP 6.5 working conference)*, North Holland, Nottingham, May 1984.

MUL87.

Sape J. Mullender (editor), *The Amoeba distributed operating system: Selected papers 1984-1987*, CWI Tract, Amsterdam, 1987.

NAV85.

S.B. Navathe and S. Ceri, "A Comprehensive Approach to Fragmentation and Allocation of Data in Distributed Databases," *Tutorial: Distributed Database Management*, IEEE Computer Press, 1985.

ONIO87.

J. Onions, S. Kille and P. Cockcroft, "The Componant Parts of the PP Mail System," *Internal Report*, University of Nottingham/University College London, UK, January 1987.

ONIO88.

J. Onions and M.T. Rose, "The Applications Cookbook," *Proceedings of the IFIP 6.5 Working Conference 1988*, North Holland, October 1988. (to be published)

OPP81.

Derek C. Oppen and Yogen K. Dalal, *The Clearinghouse: A Decentralised Agent for Locating Named Objects in a Distributed Environment*, Xerox Office Products Division, Palo Alto, USA, July 1981.

PANK87.

Uta Pankoke-Babatz, "Requirements for Group Communication Support in Electronic Communication," *IFIP 6.5 International Working Conference on Message Handling Systems*, North Holland, Munich, April 1987.

PIR74.

Robert M. Pirsig, *Zen and the Art of Motorcycle Maintenance*, Corgi Books, London, 1974.

PREV88.

J. Prevost, "High Bandwidth Services and ISDN," *Research into Networks and Distributed Applications (proceedings of EUTECO 88)*, North Holland, April 1988.

PRIN87.

Wolfgang Prinz and Rolf Speth, "Group Communication and Related Aspects in Office Automation," *Proceedings of the 1987 IFIP 6.5 working conference on Message Handling Systems*, North Holland, April 1987.

ROS78.

D. J. Rosenkrantz, R.E. Steams and P.M. Lewis II, "System Level Concurrency Control for Distributed Database Systems," *ACM TODS*, vol. 4, no. 2, June 1978.

ROSE88.

M.T. Rose, *The ISODE Development Environment: User's Manual*, The Wollongong Group, Palo Alto, CA, USA, February 1988. Version 4.0

ROT77.

J.B. Rothnie Jr. and N. Goodman, "A Survey of Research and Development in Distributed Database Management," *Proceedings of the Third International Conference on Very Large Databases*, October 1977.

ROWE86.

Lawrence A. Rowe and Michael Stonebraker, "The Commercial Ingres Epilogue," *The Ingres Papers*, pp. 63-82, Addison Wesley, 1986.

RTI85.

Introduction to Ingres, Relational Technology Inc., Alameda, CA, USA, January 1985.

SALT78.

J.H Saltzer, "Naming and Binding Objects," *Lecture Notes in Computer Science 60*, pp. 99-208, Springer-Verlag, 1978.

SANT86.

Horst Santo and Michael Tschichholz (editors), *VERDI: A Distributed Directory System for the German Research Network*, Gesellschaft fur Mathematik und Datenverarbeitung (GMD), Bonn, West Germany, July 1986.

SHOC78.

John F. Shoch, *Inter-Network Naming, Addressing and Routing*, Xerox Palo Alto Research Center, Palo Alto, USA, 1978.

SIR86.

F. Sirovich, *Access Control Model and Large Organisations*, System Wizard SRL, Ivres, June 1986. Issued as part of the THORN project

SIRB84.

Marvin. A Sirbu Jr. and Juliet B. Sutherland, "Naming and Directory Issues in Message Transfer Systems," *Computer Based Message Services (proceedings of the 1984 IFIP 6.5 working conference)*, North Holland, May 1984.

SMIT88.

Hugh T. Smith, "The Requirements for Group Communication Services," *Research into Networks and Distributed Applications (proceedings of the EUTECO 88 conference)*, North Holland, April 1988.

SOLL85.

Karen Rosin Sollins, *Distributed Name Management*, Massachusetts Institute of Technology, 1985. PhD Thesis: MIT/LCS/TR-331

SOLL87.

Karen R. Sollins and David D. Clark, "Distributed Name Management," *Proceedings of the 1987 IFIP 6.5 International Working Conference on Message Handling Systems*, North Holland, April 1987.

SPET88.

R. Speth, "Cost 11TER - Overview About the Work," *Research into Networks and Distributed Applications (Proceedings of EUTECO 88)*, North Holland, April 1988.

STEF88.

J.B. Stefani, "Communication and Object Management in Distributed Office Information Systems: Objects and Basic Services," *Research into Networks and Distributed Applications (proceedings of EUTECO 88)*, North Holland, 1988.

STON85.

M. R. Stonebraker, E. Wong, P. Kreps and G. Held, "The Design and Implementation of Ingres," *The Ingres Papers*, pp. 5-45, Addison Wesley, 1985.

TAN87.

Andrew S. Tannenbaum, *Operating Systems: Design and Implementation*, Prentice-Hall, 1987.

TER84a.

Douglas B. Terry, Mark Painter, David W. Riggle and Songnian Zhou, *The Berkeley Internet Domain Name Server*, University of California, Berkeley, USA, 1984.

TER84b.

Douglas B. Terry, *An Analysis of Naming Conventions for Distributed Computer Systems*, Computer Science Division, UCLA, Berkeley, CA, USA, 1984.

WAL85.

A. Walker, *The UNIX Environment*, John Wiley and Sons, 1985.

WEIS88.

Karl-Heinz Weiss, *Private Correspondance*, GMD-Fokus, Berlin, 1988.

WHIT84.

J.E. White, "A User-Friendly Naming Convention for Use in Communication Networks," *Computer-Based Message Services (Proceedings of the 1984 IFIP 6.5 working conference)*, North Holland, Nottingham, May 1984.

WILB88.

Sylvia B. Wilbur and Robert E. Young, "The COSMOS Project: A Multi-Disciplinary Approach to Design for Computer-Supported Group Working," *Research into Networks and Distributed Applications (proceedings of the EUTECO 88 conference)*, North Holland, April 1988.

WILS88.

Paul Wilson, "Key Research in Computer Supported Cooperative Work (CSCW)," *Research into Networks and Distributed Applications (proceedings of EUTECO 88)*, North Holland, April 1988.

ZIM88.

Wolfgang Zimmer, "Network Management Activities within IFIP TC 6," *Research into Networks and Distributed Applications (proceedings of EUTECO 88)*, North Holland, 1988.

ZLO75.

M. Zloof, *Query by Example*, 44, Proceedings NCC, May 1975.