

ANSAware and DCE - A Comparison

Phil Adcock, Gordon S. Blair, David Hutchinson
Computing Department
Lancaster University
Lancaster LA1 4YR
UK

May 1994

email : (pha, gordon, dh)@comp.lancs.ac.uk

Abstract

The aim of this document is to survey current work in distributed systems architectures with special emphasis on *integrative standards*. Two architectures will be considered in detail. The first is ANSAware [APM,93b] which is an implementation of the ANSA (Advanced Networked Systems Architecture) reference model from APM and which has contributed to ISO standards for Open Distributed Processing (ISO/ODP) [ISO,92]. The second is the Distributed Computing Environment (DCE) [OSF,92] from the Open Software Foundation. Related work on the Common Object Request Broker (CORBA) architecture [OMG,91] from the Object Management Group (OMG) are also considered where appropriate. The report is structured as follows. Chapter 1 overviews how current standards in distributed computing and networking have evolved up to the present day. Chapter 2 discusses the features and functionality of ANSAware in the context of ODP. Chapter 3 then provides a similar discussion related to DCE. Following this, chapter 4 compares the features provided by DCE and ANSAware and highlights the advantages and disadvantages of each. Finally, chapter 5 presents some concluding remarks.

Table of Contents

Chapter 1 - Integrative Standards	1
1.1 Introduction - The Evolution of Networking Standards	1
1.2 The Emerging Integrative Standards	1
1.2.1 ISO Open Distributed Processing (ODP)	2
1.2.1.1 The Descriptive Model of ODP	2
1.2.1.2 The Prescriptive Model of ODP	3
1.2.1.3 ODP Object Model	3
1.2.2 The Open Software Foundation (OSF)	4
1.2.3 The Object Management Group (OMG)	4
1.3 Summary	5
Chapter 2 - ANSAware	6
2.1 Introduction	6
2.2 Computational Model	6
2.2.1 Object Model	6
2.2.1.1 General Concepts	6
2.2.1.2 Templates	7
2.2.1.3 Types and Subtypes	7
2.2.1.4 Classes and Subclasses	7
2.2.2 Transparencies	8
2.2.3 Trading and Invocation	8
2.3 Engineering Model	9
2.3.1 The Nucleus	9
2.3.2 Interfaces and Engineering Objects	10
2.3.3 Communications	10
2.3.4 Sessions	10
2.3.5 Execution Protocols	11
2.3.5.1 The Remote Execution Protocol (REX)	11
2.3.5.2 The Group Execution Protocol (GEX)	12
2.3.5.3 The Message Passing Service (MPS)	13
2.4 Technology Model	14
2.5 ODP Functions	14
2.5.1 Trading	14
2.5.2 Factories	16
2.5.2.1 Functionality	16
2.5.2.2 Implementation	17
2.5.3 Node Management	18
2.5.3.1 Functionality	18
2.5.3.2 Persistence and Migration	18
2.5.4 Administration	19
2.6 Programming in ANSAware	19
2.6.1 The Purpose of this Section	19
2.6.2 IDL File	19
2.6.3 Server Code	19
2.6.3.1 Prepc Statements	20
2.6.3.2 The Body Procedure	20
2.6.3.3 Implementing Interface Operations	21
2.6.4 Client Code	21
2.7 Summary	22
Chapter 3 - DCE	23
3.1 Introduction	23
3.2 An Overview of DCE	23
3.3 The Fundamental Services	24
3.3.1 The Threads Service	24
3.3.2 Threads Implementation	25

3.3.3	The Remote Procedure Call Service.....	25
3.3.4	Remote Procedure Call Implementation.....	27
3.3.5	The Directory Service	29
3.3.6	Directory Service Implementation	30
3.3.6.1	The Cell Directory Service.....	30
3.3.6.2	The Global Directory Service.....	31
3.3.7	The Time Service	31
3.3.8	Time Service Implementation	32
3.3.9	The Security Service	34
3.3.10	Security Service Implementation	35
3.4	Data Sharing Services	37
3.4.1	The File Service.....	37
3.4.2	File Service Implementation.....	37
3.4.3	The Diskless Support Service.....	38
3.4.4	Diskless Support Implementation.....	39
3.5	Programming in DCE.....	39
3.5.1	Purpose of This Section	39
3.5.2	IDL	39
3.5.3	Server Code	40
3.5.3.1	Implementing Operations.....	40
3.5.3.2	Server Initialisation	41
3.5.4	Client Code.....	43
3.6	Summary.....	43
Chapter 4	- ANSAware vs DCE	44
4.1	Introduction.....	44
4.2	Architectural Issues	44
4.2.1	Motivations Behind the Architectures	44
4.2.2	From Client-Server to Object-Based	44
4.2.3	More on Integrative Standards	44
4.2.3	ODP Viewpoints.....	45
4.3	Properties and Services.....	45
4.3.1	What do you need from a distributed system?	45
4.3.1.1	Some Definitions - Properties	46
4.3.1.2	Some Definitions - Services	46
4.3.2	Properties.....	47
4.3.2.1	Global Names	47
4.3.2.2	Global Access	47
4.3.2.3	Global Security.....	48
4.3.2.4	Global Management.....	48
4.3.2.5	Global Availability.....	49
4.3.3	Services.....	49
4.4	Programming Issues.....	51
4.4.1	Comparing IDL's.....	51
4.4.2	Comparing Application Code	51
4.5	Summary.....	52
Chapter 5	- Concluding Remarks	53
5.1	Introduction.....	53
5.2	Major Results	53
5.3	Other Results	53
5.4	Looking Ahead	54
5.4.1	Achieving Convergence.....	54
5.4.2	New challenges	55
5.4.2.1	Multimedia	55
5.4.2.2	Computer Supported Cooperative Working	56
5.4.2.3	Mobility	57
5.4.2.4	Future Work	58
5.5	Summary.....	59

References	60
------------------	----

Chapter 1 - Integrative Standards

1.1 Introduction - The Evolution of Networking Standards

In 1984 the Open Systems Interconnection Reference Model (OSI/RM) [ISO7498,1984] became an international standard providing a multi-layered networking architecture. This architecture prescribes a framework within which networking standards can be developed. The OSI/RM facilitates the task of defining standards to link heterogeneous computers by separating protocol functionality into seven layers, each with a small amount of service primitives. The *physical*, *data link* and *network* layers provide network specific facilities and are separated from the higher layers by a *transport* layer which uses the layers below it to provide a universal transport service. The upper three layers of the architecture, the *session*, *presentation* and *application* layers, provide a standard set of services such as data encryption, synchronisation and remote procedure calls (RPC) and access the lower layers of the architecture via the transport layer.

Subsequent research has identified the potential benefits of distributed computing architectures, and the need for a unified standard in distributed computing systems. This has led to the ongoing development of work on standards integration. The term *integrative standards* (discussed in section 1.2) denotes the process of integrating services into an overall framework.

1.2 The Emerging Integrative Standards

The emerging *integrative distributed systems standards* aim to define an open framework for distributed systems. Integrative standards consider standards *within* end systems to allow the full functionality of a distributed system to be described. The broad range of issues encompassed by integrative standards distinguish them from the OSI/RM which is primarily concerned with communication *between* end systems (figure 1.0). Integrative standards allow us to place narrower but related standards in context with the broader issues of distributed systems. The MPEG standard for video compression [Liou,91], [Gall,91] is one example.

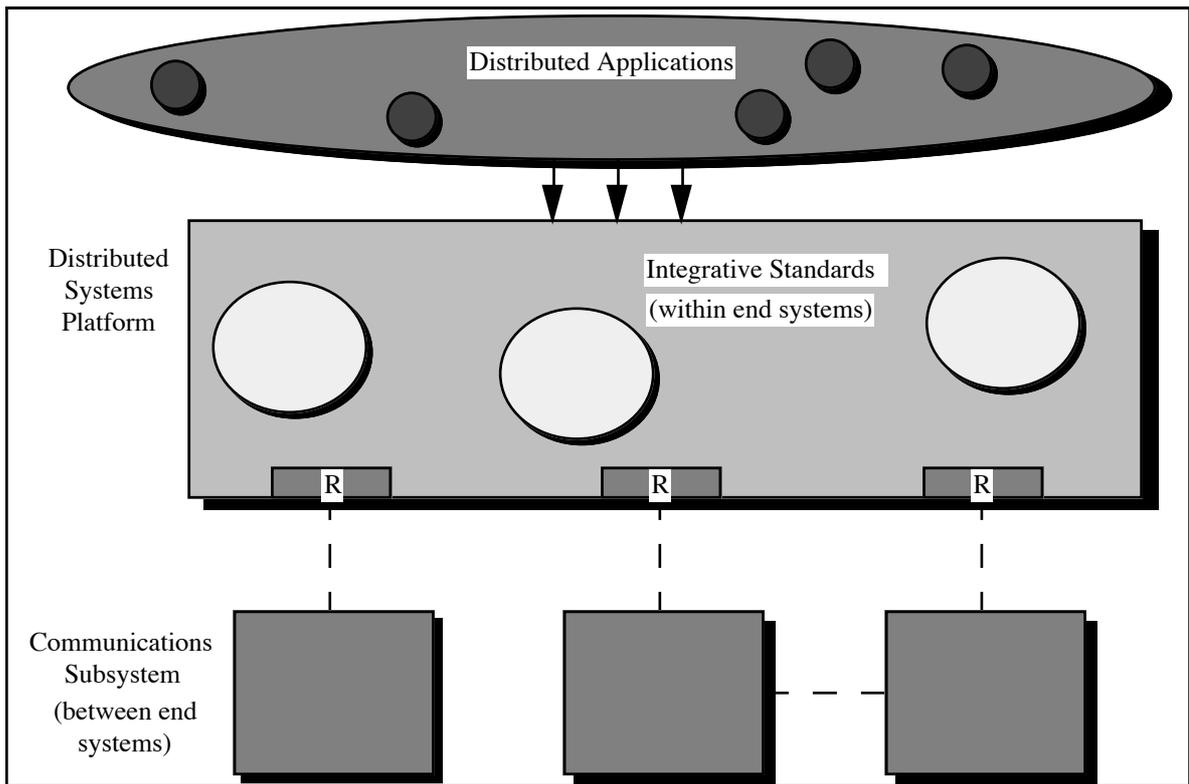


Figure 1.0 - Integrative standards and the OSI/RM

At present there are three potential integrative standards: the Open Distributed Processing standard of the ISO [ISO,92], the Distributed Computing Environment (DCE) of the Open Software Foundation [OSF,92] and standards endorsed by the Object Management Group [OMG,92].

Note that integrative standards in general may be classified into two categories:-

i) de jure

Integrative standards in this category have been published by an official standards body such as ISO/IEC. De jure standards are either in *draft* stage or have been *accepted*. Draft standards become accepted over a period of time via a series of ballots by various international and national standards bodies. An example of a de jure standard is OSI which is an accepted international standard.

ii) de facto

These standards are widely adopted but have not been officially recognised by an official standards body. Examples of de facto standards include the MSDOS operating system for personal computers from Microsoft, the SNA protocol for communication with IBM systems and the TCP/IP protocol suite.

Note that ODP is a de jure standard while OSF and OMG are de facto standards. In the following sections, the three major integrative standards are further described:-

1.2.1 ISO Open Distributed Processing (ODP)

The ISO established a new work program to define a Reference Model for Open Distributed Processing (RM-ODP). The intention is to provide common definitions of concepts and terms for distributed processing, a generalised model of distributed processing using these concepts and terms, and a general framework for identifying and relating together open

distributed processing standards. This latter point applies particularly to the integration of the OSI communications standards into the wider ODP framework.

ODP relates to the upper layers (layers 5 - 7) of the OSI/RM and provides a *descriptive* model and a *prescriptive* model for referring to open distributed systems.

1.2.1.1 The Descriptive Model of ODP

The descriptive model provides a common vocabulary and a consistent view of distributed systems by describing systems in terms of *viewpoints*, each of which is a complete and self contained perspective on distributed systems in a terminology appropriate to a particular interested party. The five viewpoints are *enterprise*, *information*, *computational*, *engineering* and *technology*. The following is a brief summary of the five viewpoints of ODP:-

enterprise viewpoint

Describes a system in terms of the environment in which it operates, the people that use the system and those that own it. This places the system in the context of the enterprise it serves. The enterprise model is concerned with organisation of resources and people and the interface that the distributed system presents to the outside world.

information viewpoint

Models the information flowing through a distributed system. This model does not differentiate between tasks which are performed manually and those which are automated. This viewpoint is used by business analysts and information engineers/managers to identify the rules and constraints governing information flow and modelling how data is to be represented by the system.

computational viewpoint

Describes the system from the point of view of a systems programmer. This model is concerned with programming abstractions such as functions and data types and identifying distribution transparencies. It is not concerned with elements of the underlying system which are required to support the computational interface.

engineering viewpoint

Describes the underlying support environment of the architecture. This includes the implementation of transparency mechanisms and the code necessary to support a computational environment over particular operating systems and networks. Characteristics such as performance, quality of service, and reliability are dealt with at this level.

technology viewpoint

Describes the underlying system in terms of the hardware and software which comprise it, including configuration, installation and maintenance. Systems with different operating systems have their own technological viewpoint.

The two most relevant viewpoints for describing a distributed architecture in terms of system programming and the construction of the architecture are the computational and engineering viewpoints.

1.2.1.2 The Prescriptive Model of ODP

The prescriptive model constrains the descriptive model by prescribing the technologies which support distributed processing to those which support open distributed processing. This model contains five prescriptive languages based on the five viewpoints detailed in the

descriptive model. The prescriptive model also prescribes the ODP *functions* (described in section 2.5) which are required to support open distributed processing. The engineering language defines reference points for ODP functions and therefore the placement and interrelationship of these is defined in this language. □

1.2.1.3 ODP Object Model

The ODP computational viewpoint specifies an *object model* which describes the components of a distributed system as a set of potentially distributed objects. The services that a particular object offers are described in an *interface* which separates the services provided from the underlying implementation of the object (which may vary from system to system). Information local to an object is said to be the *encapsulated state* of that object. Encapsulated state may only be accessed through the *invocation* of operations specified in the interface. Offering services through a well-defined interface abstracts over distribution and places emphasis on behaviour (i.e. invoking an operation on an interface). This provides decoupling between objects which is important in terms of a system of potentially distributed objects. In order to invoke operations, the service provider must *export* its interface to a trader (a directory of available services) and the service requester must *import* the service to create a *binding* (figure 2.1). These concepts will be described in chapter 2.

ODP considers objects to be each of the following:-

- *unit of service* - due to the modular nature of objects.
- *unit of distribution* - objects may be relocated.
- *unit of failure* - a whole object fails or none of it does.
- *unit of replication* - to provide higher availability.
- *unit of security* - for access control.

In summary, the object model is a fundamental concept of ODP. Objects provide a method of abstraction over the entities populating a distributed system. They combine the power of object-orientation with the simplicity of abstraction. Further details of objects can be found in chapter 2.

1.2.2 The Open Software Foundation (OSF)

DISCLAIMER: ALL OPINIONS EXPRESSED IN THE FOLLOWING TEXT ARE THOSE OF THE AUTHOR AND IN NO WAY SHOULD BE TAKEN AS A STATEMENT OF POSITION FROM THE RELEVANT STANDARDS BODY

OSF is an industry funded non-profit making organisation sponsored by a number of leading information technology manufacturers including IBM, Hewlett-Packard, Digital, Siemens-Nixdorf, Hitachi and Bull. The OSF standards offer openness through a process of the selection, certification and integration of software technologies which guarantee stability and continuity to customers. For example, OSF have adopted the Mach micro-kernel and the UNIX SVR4 interface as the OSF/1 operating system standard.

The OSF's Distributed Computing Environment (OSF/DCE) [OSF,92] is an operating system independent and network independent software platform intended to support the development of distributed applications in an open environment. DCE applications are built in terms of multi-threaded objects with abstract data type interfaces which communicate via remote procedure calls (RPC). Standard services provided with DCE include RPC, threads, directory, time and authentication services.

DCE falls naturally into the engineering viewpoint of the ODP framework described above. The platform incorporates many existing standards and has the backing of a weighty commercial consortium. In terms of existing standards, DCE incorporates X.500 directory services, X/Open transport interfaces and the ISO's ROSE, ACSE, session and presentation services. Earlier OSF products such as the OSF/1 operating system and the OSF-Motif user interface toolkit have already achieved significant commercial impact in their fields. DCE seems likely to make a similarly large impact in the field of open distributed platforms.

1.2.3 The Object Management Group (OMG)

DISCLAIMER: ALL OPINIONS EXPRESSED IN THE FOLLOWING TEXT ARE THOSE OF THE AUTHOR AND IN NO WAY SHOULD BE TAKEN AS A STATEMENT OF POSITION FROM THE RELEVANT STANDARDS BODY

The OMG is another industry backed consortium which aims to integrate existing standards into a comprehensive distributed open systems framework. Participants include DEC, Hewlett-Packard, NCR and SunSoft. The OMG explicitly advocates the use of object-orientation for the achievement of its goals and has defined an object model and architecture. Services offered by OMG include:-

- *a naming service* - associates unique names with services.
- *an event service* - for handling asynchronous events between objects.
- *life cycle services* - protocols, services and conventions for object manipulation. The unit of manipulation is a graph of associated objects. Typical services include creating, removing, copying and moving objects.
- *an association service* - for creating associations between objects across heterogeneous systems. The objects themselves may have no knowledge that they are associated.
- *a persistence service* - manages the persistent state of an object.

The most tangible offering of the OMG to date is the *common object request broker* (CORBA) [OMG,91] which provides functionality comparable to OSF's DCE. Although CORBA is the only object request broker currently defined, multiple ORBs can exist in principle.

As defined by the OMG, CORBA provides the mechanisms by which objects transparently make requests and receive responses. It provides interoperability between applications on different machines and seamlessly interconnects multiple object systems. The role of the service database in CORBA is taken by objects called *Interface Repositories*. *Object Adapters* are also provided which offer generic infrastructure services such as data persistence (activation and deactivation), security of interactions and generation and interpretation of object references. Interfaces in CORBA are defined in a notation called IDL (interface definition language). This is a subset of ANSI C++ which, however, does not contain any algorithmic structures or variables.

The definition of the CORBA is purely object-oriented: it is defined by its interfaces and not by its implementation. Thus CORBA itself is defined in IDL and each ORB implementation makes available a standard IDL file known as orb.idl.

1.3 Summary

This chapter has introduced the concept of integrative standards and has placed these in context with other existing standards. Three integrative standards, OSF, OMG and ODP have been discussed and the fundamental concepts behind these standards explained. The remaining chapters of this document will draw on the integrative standards discussed here as

points of reference. The structure of the remainder of this document is as follows. Chapter 2 discusses the features and functionality of ANSAware in the context of ODP. Chapter 3 then provides a similar discussion related to DCE. Following this, chapter 4 compares the features provided by DCE and ANSAware and highlights the advantages and disadvantages of each. Finally, chapter 5 presents some concluding remarks.

Chapter 2 - ANSAware

2.1 Introduction

The ANSA reference model is an example of a framework for creating distributed computer systems. Between 1984 and 1988, the ANSA project was supported by the UK Alvey research programme and then became part of the European ESPRIT Integrated Systems Architecture (ISA) project until 1993. The ANSA project has now entered an industry supported stage known as *Phase 3*. This work has contributed significantly to ODP in its formative years and continues to be a major influence on standards.

The ANSA project aims to provide an architecture for distributed systems which satisfies the following properties:-

- *genericity* - ANSA is generic to many fields of application.
- *state of the art* - The technical content is innovative and up-to-date.
- *portability* - The architecture is portable across a wide range of computer and network topologies, operating systems and programming languages.
- *heterogeneity* - ANSA is operable in heterogeneous multi-vendor environments.
- *modular* - The architecture is modular in structure.
- *distribution policies* - ANSA supports a wide range of distribution policies e.g. for fault handling.
- *application programmer oriented* - ANSA is oriented towards the requirements of application programmers.
- *internetworking* - ANSA provides for internetworking between autonomously managed networks.

The key concepts of the ANSA reference model have been implemented in the form of the ANSAware software suite. ANSAware provides the user with a set of distributed system services which abstract away from the underlying and potentially heterogeneous infrastructure. The primary goal of ANSAware is to simplify application development and maintenance for distributed systems.

This section will describe the ANSA reference model in terms of the computational, engineering and technology viewpoints of ODP. **References to the ANSAware software suite refer to the current release of ANSAware at the time of writing (version 4.1).**

2.2 Computational Model

2.2.1 Object Model

This section discusses the key concepts of the ANSA object model. The object model is discussed under four main headings; general concepts, templates, types and classes.

2.2.1.1 General Concepts

The ANSA computational model is built on the concept of providing a *service*. ANSA

consists of interactions between service users (clients) and service providers (servers) where servers may also act as clients to other servers. Services may be separated into *application* services which are specific to a particular task, such as a booking service for a theatre booking system, and *architectural* services which are generic to a wide range of tasks, such as the ANSAware trader (described in section 2.2.3). Services are provided at an *interface* which is the unit of service provision in ANSAware and consists of a list of zero or more *operations*. Each operation has a unique signature and set of attributes.

A programmer may define customised data types in interface definitions. Such data types provide a language and system independent view of services and thus enable data transfer between heterogeneous systems. Each interface is identified by a data structure known as an *interface reference*. Interface references are treated as first class entities in ANSAware, i.e. they are distinct entities that can be offered by an object as opposed to merely being a part of an object.

In terms of an object-based paradigm, ANSAware describes a distributed system in terms of potentially distributed *objects* with defined interfaces through which the services of an object are accessed. Objects are a collection of zero or more interfaces which means that a particular object may be described by several interfaces, with different operations and data types. An object described in terms of the services that it offers is known as a *computational object*.

Objects may be grouped into templates, types and classes. The function of each of these is described in the following three sections.

2.2.1.2 Templates

Templates¹ describe objects in terms of their common features e.g. state and operations. Objects with the same common features may be described by the same template. Furthermore, objects may be generated from templates. This process is known as *instantiation*.

Several objects instantiated from different templates may share many similarities. It is useful to be able to classify objects independently of their templates. This is the function of a *type*.

2.2.1.3 Types and Subtypes

A type is a predicate which may be applied over a set of objects. A type may refer to attributes of an object, or operations on an object. An object is of a specific type if it satisfies the type predicate. For example, a type predicate for a square will specify that all sides of the square must be equal. This is an attribute-based predicate.

Objects may be classed into a subtype hierarchy where *type 'X'* is a subtype of *type 'Y'* if *type 'X'* satisfies all the properties of *type 'Y'*. Thus, a subtype hierarchy represents an implication between types.

2.2.1.4 Classes and Subclasses

All objects satisfying a particular type are grouped into a class. Thus, there is a class associated with every type. Classes and types provide different abstractions for looking at objects. We may either start with a collection of objects (i.e. a class), or define a predicate (i.e. a type) and determine which objects satisfy it.

Objects may be grouped into a *subclass hierarchy* which corresponds to an equivalent subtype hierarchy.

¹The reader is referred to [Rudkin,93] for further information on templates, types and classes.

2.2.2 Transparencies

The computational model provides *transparencies* in order to mask the complexities of the underlying distributed system. Typical distribution transparencies include:-

- *access* - hides whether objects are local or remote and allows objects to interact uniformly over heterogeneous systems.
- *location* - a client may interact with a server without knowing where the server is physically located.
- *replication* - multiple instances of an interface may be treated as one, without considering the organisation of the supporting objects.
- *migration* - objects may move from location to location without the knowledge of users.
- *concurrency* - the concurrent execution of code is hidden.
- *atomicity* - either all of a set of actions are performed or none are supporting transaction processing and failures.

ANSAware currently implements access, location and replication transparencies. Concurrency and atomicity transparencies are also available in ANSAware version 4.1 as an option. These are implemented via the Arjuna transaction service [Shrivastava,91]. Migration transparency will also be implemented in the near future. A programmer may choose selectively which transparencies should be used in a particular context.

2.2.3 Trading and Invocation

Objects *export* their services to a 'database' of services available in the system known as a *trader*. An exported service is an *offer*. A client object may then *import* services advertised in the trader creating a *binding* between a client object and a server object. A client then may then *invoke* operations in the server (figure 2.1).

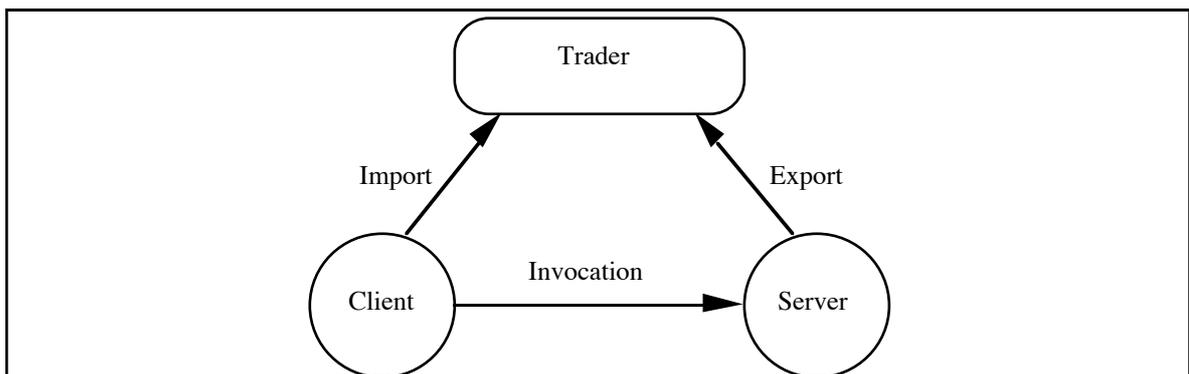


Figure 2.1 - the ANSA object model

An entry in the trader consists of an abstract data type signature for an object together with a set of attributes which are associated with the object. When importing an object, clients may specify a set of requirements in terms of operations and attribute values. A suitable candidate in the trader is then selected according to these criteria. This is a 'yellow pages' style of approach (as discussed further in section 2.5.1). It should be noted that when searching for a suitable candidate it is not necessary to find an exact match. Any object

which provides at least the required functionality may be used, through a subtyping policy supported by ANSAware. Once a suitable match is made, a binding is established between the client and server objects and invocations may take place.

2.3 Engineering Model

2.3.1 The Nucleus

The ANSAware engineering model supports the facilities provided by the computational model, including support for distribution transparencies.

At the lowest level of abstraction ANSAware runs on a *node* which may be a computer, or a process on a machine. A *nucleus* sits on top of a node and takes the resources available in the node and provides a uniform, platform-independent, distributed computing environment. The nucleus may be considered a resource manager and generally allocates resources to *capsules*. Capsules are a collection of zero or more objects and are a unit of autonomous operation, each with a separate address space.

The nucleus provides a capsule with facilities to create new capsules, with persistence of state across interactions (but not failures), with inter-capsule communication facilities and with encapsulation (i.e. a capsule provides a single protected address space). Engineering objects may be linked together to form a *template* for a capsule.

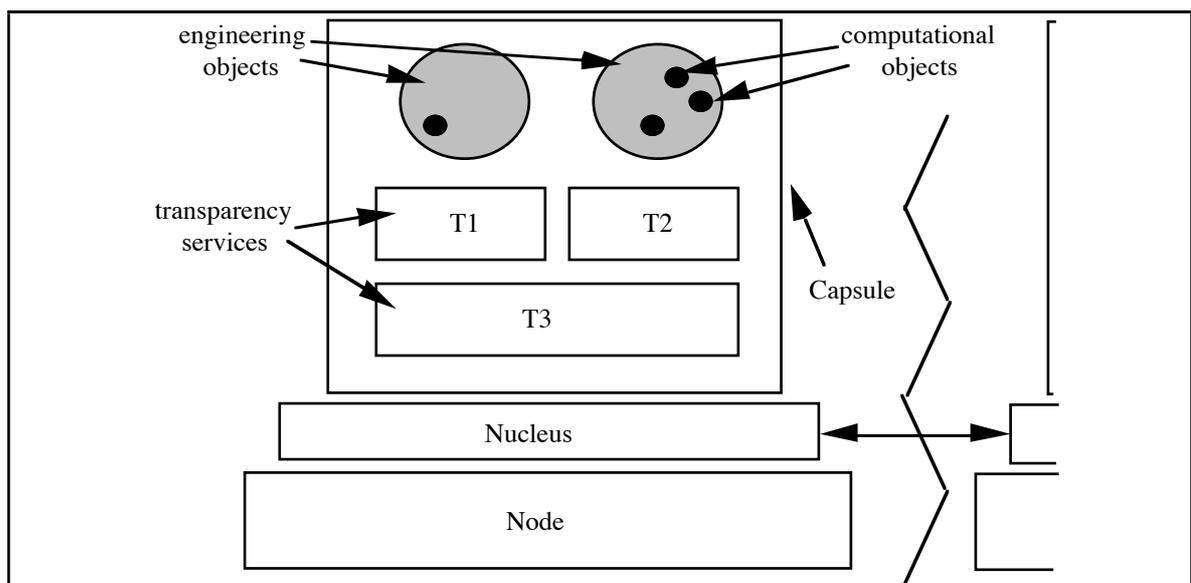


Figure 2.2 - the ANSA engineering model

The nucleus provides capsules with a number of resources. Firstly, the nucleus provides the concept of *threads* which are an independent flow of control through a capsule. Threads are allocated to *tasks* (a virtual processor which provides a thread with the resources it needs) and are retained by the task until the thread terminates. In terms of allocating the resources of a capsule, a task is the unit of execution and may be executed on one or more processors. Threads may be *synchronised* using eventcounts and sequencers [Reed,79].

The nucleus provides the *socket* as the unit of addressing for inter-capsule invocations. Sockets are generally encapsulated in interface references (which are provided through the binder) and are the server end of a communication. *Plugs* are the access point for the client

of an interface. Plugs at the client end of a communication map to sockets at the server end of a communication and communicate through a *channel* (the path from a plug to a socket).

Other facilities provided the nucleus include *sessions* which provide a cache for channel information, synchronisation and resource management functions in a session table. A local *binder* service in each capsule performs binding functions such as creating interface references.

Facilities such as *trading*, *factories*, *node management*, *notification services* and *administration* are provided at the engineering level. These facilities fall into the category of ODP functions and will be discussed in section 2.5. Finally, the nucleus services are accessed via the *interpreter*, the interface to the nucleus.

2.3.2 Interfaces and Engineering Objects

An Interface Definition Language (IDL) is provided for specifying interfaces. IDL files are compiled into stub files (for marshalling and unmarshalling) and header files (for mapping types defined in the IDL into a host language) using a stub compiler, *stbc*. The header and stub files are currently generated as C code. These files are included into the application program code and are compiled as part of the application program (for example a client or a server). Statements from a Distributed Processing Language *prepc* may be embedded in the host language. Prepc supports facilities such as importing, exporting and invocation. Compiling the source files generates one or more *engineering objects*. An engineering object is a computational object augmented by additional services to implement the various transparencies required. They are essentially the runtime representation of *one or more* computational objects. Engineering objects are the smallest objects which may be activated, passivated, distributed or migrated in ANSAware.

2.3.3 Communications

A communication protocol exists for communications between inter-nuclei communications and is defined in terms of layers, a *session service*, an *execution protocol* and a *message passing service*.

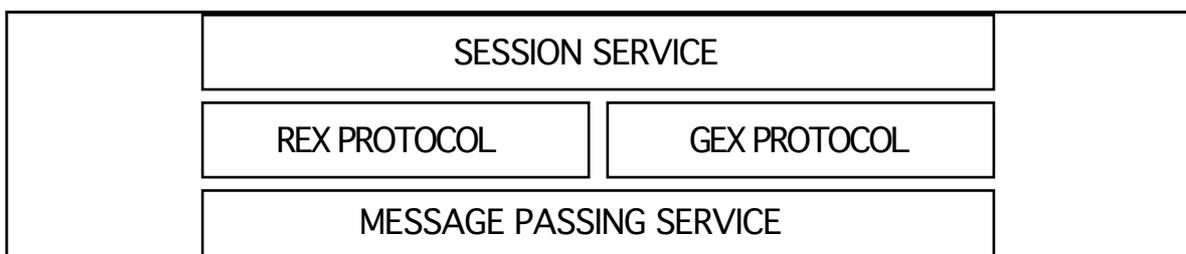


Figure 2.3 - the ANSAware communication protocol

2.3.4 Sessions

The session service manages logical sessions, i.e. a channel connection between a socket and a plug. This service provides a cache for channel information and synchronisation and resource management functions between the execution protocols and the interpreter.

2.3.5 Execution Protocols

Two execution protocols exist, a *remote execution protocol* (REX) and a *group execution protocol* (GEX).

2.3.5.1 The Remote Execution Protocol (REX)

The REX protocol is a remote procedure call protocol supporting the ANSAware computational model. It provides a mechanism supporting process interaction across a network. REX is designed to support the delivery of information between processes. This includes synchronisation and scheduling. REX is not concerned with functions such as marshalling and unmarshalling of data.

REX extends the concept of a remote procedure call in a number of ways:-

Rate Based Flow Control

Involves sending data as fast as the receiver can cope with it. The average rate of packets transmitted is adjusted as necessary. Lost packets are retransmitted at the end of a transfer. Packet loss usually occurs due to *receiver latency*. This occurs when the receiver stops listening on the network for packets while it processes the last packet arrival.

Bulk Delivery of Data

Rapid delivery of bulk data is supported by concentrating on performance optimisations. Such optimisations concentrate on minimising the number of thread switches and buffer copying operations. REX expects a client to pass its own thread and a buffer which are returned when a client-server interaction is complete. Large data packets are fragmented before transmission and are reassembled at the other end.

Asynchronous Messaging

Support is provided for both reliable synchronous calls and asynchronous casts. Calls are based on exactly once and at most once semantics. Casts are based on at most once semantics. The reliability of casts is dependent on the underlying message passing protocol [APM,93a].

The aims of REX are to provide:-

- *Low response time for calls*
- *High reliability for calls*
- *High throughput for casts*

The mechanism of a typical call is illustrated in figure 2.4. REX call interactions are based on the Birrell-Nelson design [APM,93a].

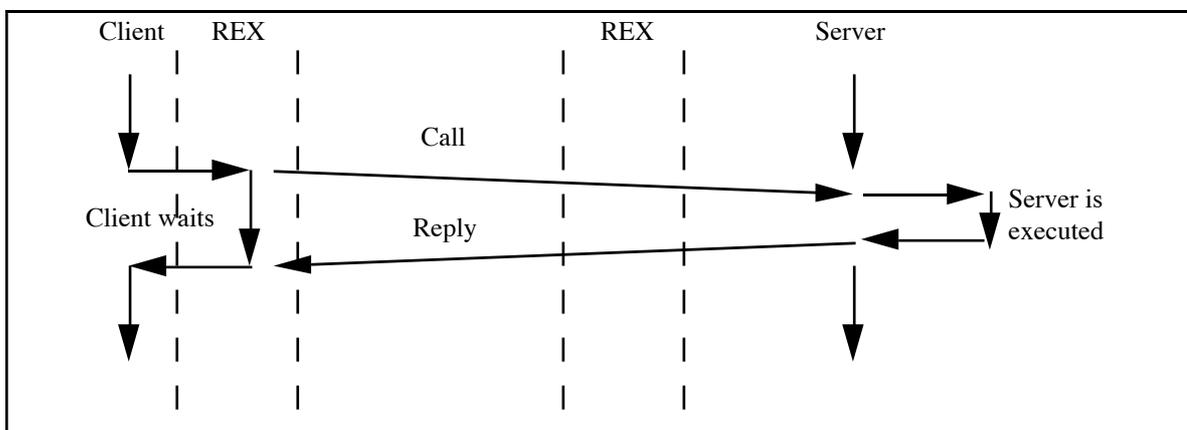


Figure 2.4 - the ANSAware call mechanism

A calling thread blocks after making a call and remains in that state until a response is received. The calling thread unblocks when a response is received. This allows the calling thread to synchronise with the response. Response messages are transmitted at regular intervals until an acknowledgement is received.

Asynchronous calls differ from synchronous calls in that they do not have to wait for a response. After a request, the calling thread only blocks long enough to transmit the message. Figure 2.5 illustrates a cast.

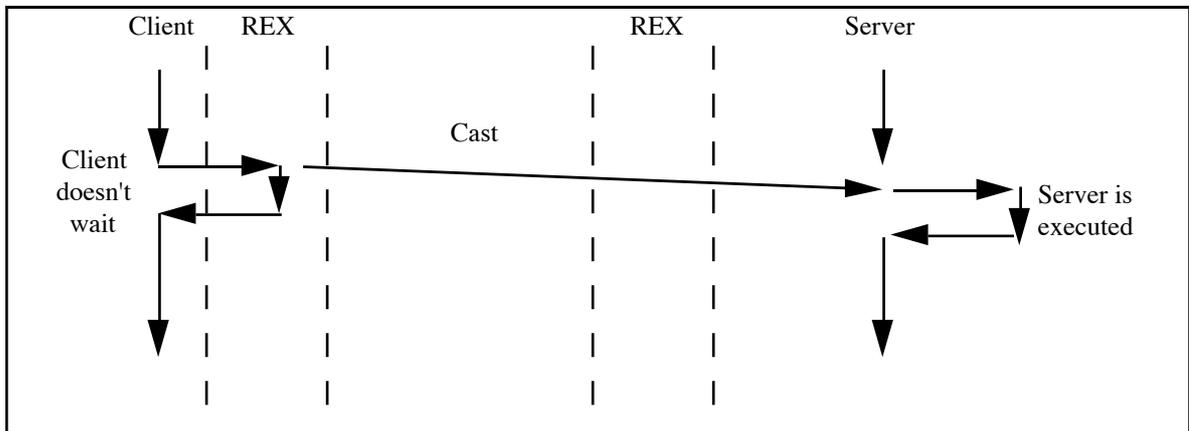


Figure 2.5 - the ANSAware cast mechanism

Casts depend on the underlying message passing service for reliability. They are not retransmitted or acknowledged.

It can be seen from the above diagrams of calls and casts that both are sequenced. Any casts that arrive out of sequence are discarded.

2.3.5.2 The Group Execution Protocol (GEX)

This protocol is a recent addition to ANSAware. It provides support for transparent active replica groups. The group execution protocol GEX provides²:-

- *Client support* - Support for transparent communications with groups including calls and casts.
- *Group member support* - Interfaces may be invoked as a member of a group. Group members may synchronise with each other.

Thus, servers may be replicated and each server may execute concurrently with other replica servers for availability. Figure 2.3 illustrates the positioning of GEX in relation to REX. GEX sits alongside REX providing similar functionality augmented by group facilities.

GEX is transparent to the trading process. If a group interface is imported, the client will bind to it by transparently using the GEX protocol. A binding will be established to each member of the group. The current implementation of GEX establishes a REX session with each group member. Replies from the group invocation are collated and returned as a single reply. Communication failures result in an error being returned to the caller.

The GEX protocol consists of several elements:-

²Note that GEX is in the early stages of development. It is likely that the specification of GEX will change in future releases of ANSAware.

Group Sequencer

Synchronises a group member with other group members. Each group member has a sequencer. Sequencers provide two distinct functions; defining an order in which invocations to groups are evaluated and managing population changes³. The current sequencer protocol establishes a logical ring of sequencers and passes a token around the ring. This establishes an evaluation order and provides a check to see if a group member has failed.

Group Relocator

Stores the state of existing groups (such as membership) and updates client references when they become out of date. The relocator also stores critical group information to allow a group to vary membership but still be invoked by clients who have stale interface references to the group.

Collator

Collates multiple messages destined for the same destination to a single message where possible. Collation may involve client requests and server responses.

2.3.5.3 The Message Passing Service (MPS)

The message passing service (MPS) handles connection-oriented and connectionless communications between nuclei. This protocol handles network functions such as routing. This will involve protocols such as the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP).

MPS is essentially an abstraction over a transport service. It provides network-level facilities which may be used by the higher layers such as REX. MPS also provides a series of optimisations. A typical example of this is an optimisation provided in the UNIX version of ANSAware. Invocations to remote procedures on the same node are passed via UNIX pipes as opposed to using either TCP or UDP. This provides more efficient communications and reduces network overhead.

Further information on the functions provided by MPS may be found in [APM,93a].

2.4 Technology Model

ANSAware is currently implemented over several operating systems some of which have different technology models, e.g. MSDOS and UNIX. ANSAware was initially prototyped over a UNIX environment. This has resulted in some bias in terms of the implementation of capsule, nucleus and application objects towards UNIX. This is explicitly recognised in the ANSAware Application Programmers Guide [APM,93]:-

“It is important to realise that UNIX was the first operating system used to prototype ANSAware and many of the implementation options made were inevitably biased towards it; however, a great deal of care has been taken to ensure that such bias is slight, and reversible, if necessary”

The technology model is concerned with mapping the ANSAware components, such as application, capsule, nucleus and engineering objects, to the underlying system. For example, in UNIX, a capsule is mapped to a process in a similar manner to VMS while a single MSDOS machine supports one capsule at a time, unless running under the Microsoft Windows environment.

³A new group member must receive all invocations to the group from the moment that it joins the group. It must not process invocations that arrived before it joined the group which may be currently processed by the group.

2.5 ODP Functions

ODP functions are the building blocks used for the construction of distributed systems from re-usable components. They are intended to overcome the problems of distribution and ensure that information processing is carried out effectively. Functions are identified in the five ODP viewpoints (predominantly the engineering viewpoint because many functions generally involve co-operation between engineering objects). Functions identified by the engineering viewpoint tend to be grouped into *transparency functions* in the computational viewpoint. An example of this is the ODP migration transparency function which relies on engineering functions such as relocation and persistence. Functions are realised as objects with a defined role, interface, and behaviour.

Some typical examples of functions include node and object management functions, coordination functions (events, groups, replication, migration), security functions and repository functions such as trading (see section 2.5.1).

2.5.1 Trading

This section discusses trading as an ODP function. Firstly the section revisits the basic concepts of trading. This is followed by a more detailed study.

The ANSAware trader provides a ‘yellow pages’ style of name service known as trading which allows the lookup of objects based on their attributes. This part of the trading function is closely linked with the type management function described below. Objects may export their interfaces to the trader which allows client objects to import interfaces offered by those objects creating a client-server binding. Exported offers may be assigned *properties* which may then be used to constrain a search for a service when importing. The trading mechanism supports the late binding of objects which allows client objects to be configured before the server objects that are required to provide services.

The ANSAware trader has a *type space* and a *context space*. Trading incorporates a *type management* scheme which manipulates the trader type space. This allows the description of interfaces and their comparison based on their attributes. All interfaces described in the trader are of a particular type and fit into a directed acyclic graph which describes all possible types supported by the trader (see figure 2.6).

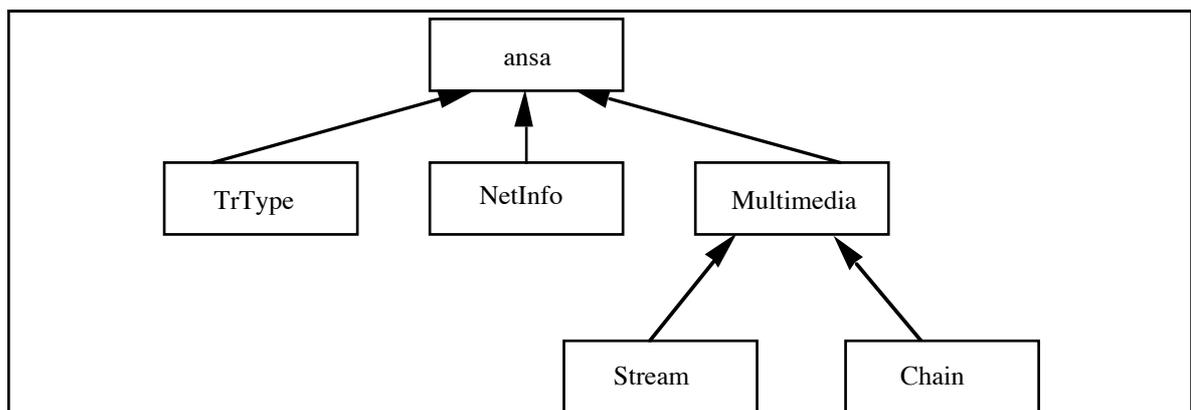


Figure 2.6 - a directed acyclic graph (DAG) representing a possible ANSAware type space

The type management functions provide the means for objects to determine the types that exist and dynamically match compatible types during the process of trading. This allows trading queries of the form ‘I would like to bind to an interface which provides me with at least this functionality’. This principle is known as *type conformance* and works on the premise that a type X conforms to a type Y if and only if type X provides at least the operations that type Y provides.

The *context space* of the trader is an organisational issue which classifies interface

instances into an administrative hierarchy as illustrated in figure 2.7.

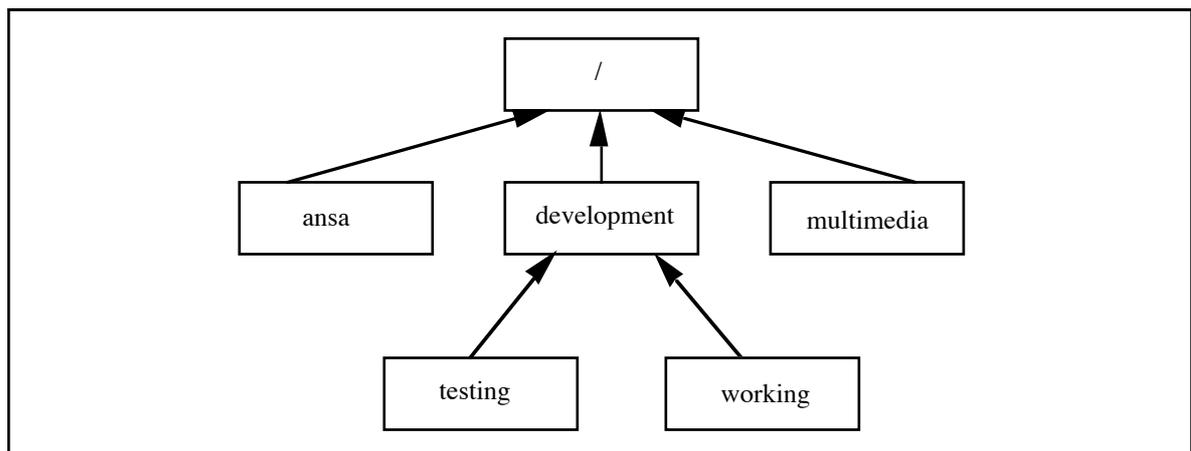


Figure 2.7 - an example ANSAware context space

Each node in the hierarchy is a *context* in the overall *context space* which is similar to the directory structure found in UNIX. When exporting interfaces, a server object must specify the context space that it wishes to export to and similarly, when importing an interface, a client must specify the context which it wishes to import from.

The trader is a *federated* object which means that traders are replicated. Federation is necessary for:-

- *Availability* - Federated traders reside on different nodes which improves the possibility of accessing the trader. Federation notwithstanding, the trader remains a critical point of failure (see following paragraph).
- *Improved Performance* - Several federated traders running in parallel on different nodes improve trader response.
- *Management* - A large namespace may be subdivided into more manageable domains. This provides a structure for the trading namespace.

Federation creates a set of *trading domains* supporting differing groups of users. Replication occurs at the level of a service class, not a service instance i.e. each replicated trader supports a different trading domain which implies that the trader is a critical point of failure. Communities of users in different trading domains may interact by placing information about their existence in the entry space of other traders. This may be achieved by binding a trader's context space into a context name in another trader's name space or by associating a remote trader with an interface instance (a proxy offer).

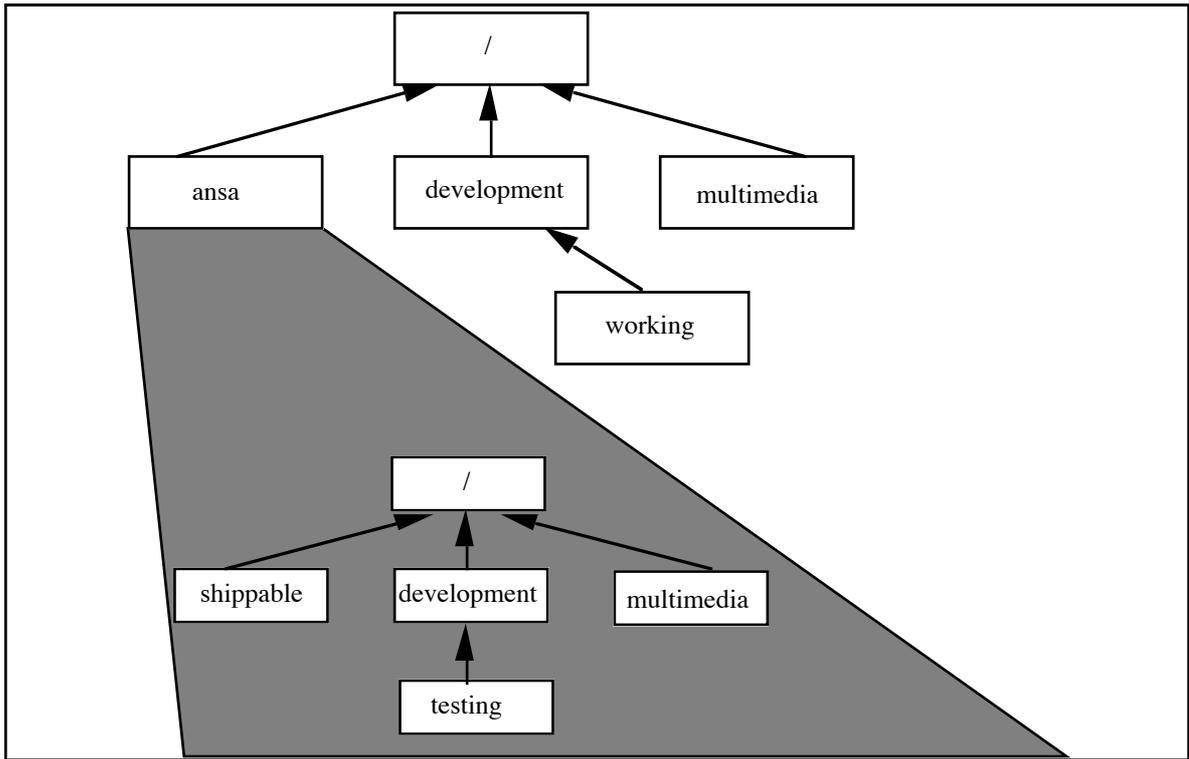


Figure 2.8 - a federated namespace

A common method of federation in ANSAware is to nominate one trader as a master trader and have other traders bind their context to the master. The above diagram represents a context space of a local trader being bound to the context name “ansa”.

2.5.2 Factories

2.5.2.1 Functionality

The factory service provided by ANSAware allows the dynamic creation of engineering objects with a desired interface type to provide particular services. In general, factories create capsules with each capsule having its own local factory service. This allows the new capsule in turn to instantiate its own engineering objects if it wishes.

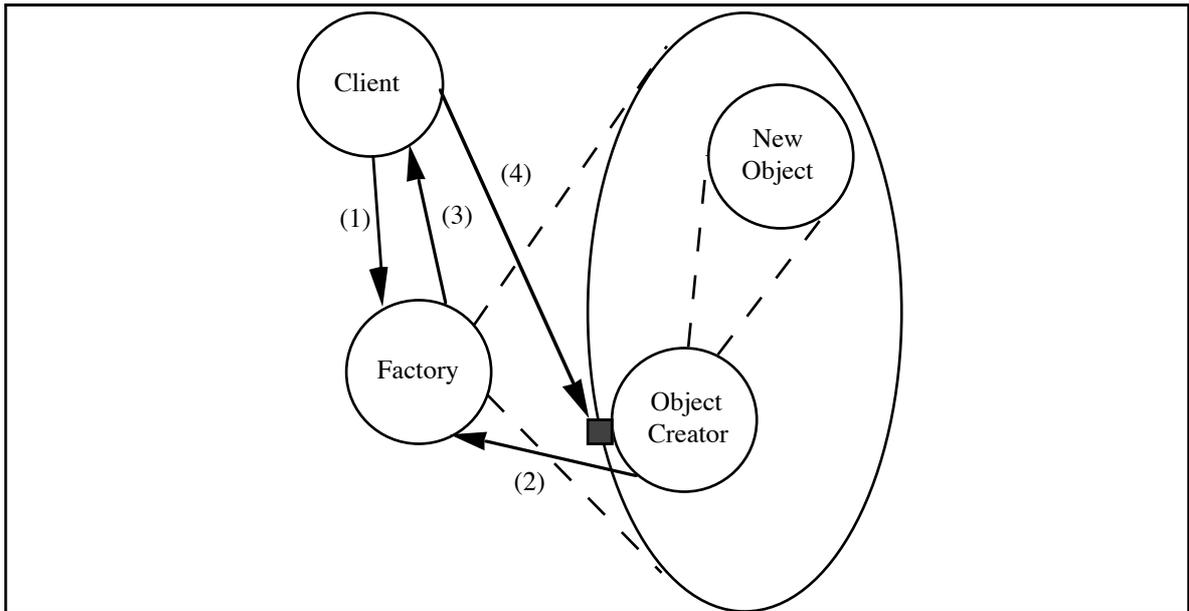


Figure 2.9 - a diagram of factory interactions

The process of creating objects is detailed in figure 2.9. A client sends an instruction to the factory to instantiate a new capsule (1). This results in a capsule which contains a single “object creator” object with a capsule interface. An interface reference to this capsule is returned to the factory (2) which passes this back to the client (3). The client may then create new objects in the capsule whenever it wishes by using an instantiate operation supported in the capsule interface (4). This operation takes a template as a parameter and returns one or more interface references to the new object depending on the type of object being instantiated.

The factory service provides two functions in addition to the creation and destruction of capsules:-

- *Capsule Monitoring* - This service advises interested parties of when a particular object is terminated i.e. destroyed. This provides a level of fault tolerance and robustness in the face of failure.
- *Capsule Existence* - Determines whether a particular capsule exists.

These functions replace the Notification Service which was used in ANSAware 3.0. The advantage of this approach is that the detection and notification of the demise of capsules without reliance on a centralised service.

2.5.2.2 Implementation

The factory service consists of a number of threads:-

- *The Grim Reaper Thread* - Scans the node on which it runs to check for the demise of capsules created by the factory service. This is achieved by scanning a list of capsules created by the factory and then interrogating the underlying operating system to see if the capsules exist.
- *Rendezvous Thread* - Checks to see if a capsule creation function has succeeded. This thread returns an error message if the rendezvous has not occurred within a specified period.
- *Factory Thread(s)* - Process invocations to the factory service interface. Multiple threads are created to handle multiple invocations.

2.5.3 Node Management

2.5.3.1 Functionality

The node manager manages the services available on a node such as the static or dynamic creation of capsules and objects and their termination. The functions of the node manager fall into three categories:-

- *Service Database*
- *Service Creation*
- *Dynamic Service Creation*

The service database provides a means for describing the services available. The database is a *persistent* database (see the following subsection 2.5.3.2). The node manager uses aliases⁴ to identify each service in the database. If aliases are specified as *permanent* then the node manager will restart the associated service if it terminates. The service will restart from its initial state. Aliases are listed in database entries along with an interface name, a context name and a properties string (for trading), a path string, an arguments string and an environment string (for capsules and factories), a capsule name (for the factory), an object template (for the capsule), and an instance limit which specifies the number of instances that can be created for this service. Operations are supplied for creating and removing entries from the service database, and viewing database entries.

Static service creation is known as *activation*. This involves creating a new instance of the service referenced by its alias using a *run* operation. If several instances of a service are activated, each is denoted by an activation number. A specific activation of a service is referenced by a combination of its alias and its activation number. Deactivation (i.e. object termination) is performed by a *kill* operation. The kill operation will terminate both permanent and non-permanent services.

Dynamic service creation is closely linked to the proxy export facility of the trader (see section 2.5.1). The dynamic service facility supports operations which add and remove proxy export offers for aliases in the trader. Once a proxy export offer is posted, all trader queries relating to the alias are routed to the node manager. The node manager will then perform an activation (or possibly return an existing one if the maximum number of activations are reached). Services created dynamically will self-terminate after a specified period of time. This time period may be specified by the programmer and defaults to ten minutes.

Both the static and dynamic creation mechanisms return an interface reference to the created services which may be used to invoke them.

The node manager uses that factory service to create and destroy objects and therefore each node needs a factory service in addition to a node manager. The node manager service can be imported from the trader just like any other ANSAware service.

2.5.3.2 Persistence and Migration

ANSAware provides limited support for *persistent objects* (i.e. objects which survive the process that created them), *passivation* and *activation*. Passivation occurs when an object is deactivated to conserve system resources. Activation refers to the process of restoring a passivated object to its state prior to passivation.

The following ANSAware facilities provide support for persistence:-

⁴In terms of a node manager, an alias is a unique string identifying a service description.

Relocator Service

This allows new interface references to be generated from stale ones (e.g. when a passivated object is activated). The relocator is described more fully in section 2.3.5.2.

Service Database

Objects may be flagged as permanent in the service database (see section 2.3.5.1). However, these objects restart from their initial state and not the state that they were in when they were passivated. This functionality must be implemented manually.

Migration is not currently supported by ANSAware. It is reasonable to assume that future support for migration (and persistence) should be provided through the node manager. This would require use of the service database and relocator facilities in conjunction with the ANSAware RPC services.

2.5.4 Administration

The administration functions provided by ANSAware include the management of trading domains, for example defining appropriate domain boundaries. Each trading domain contains a trader, one of which is designated the master trader whose function is to supply the slave traders with information regarding the location of other traders. Thus it is necessary to install, federate and maintain traders, factories and node managers. Administration support is also provided by ANSAware at the object level through management interfaces associated with an object.

2.6 Programming in ANSAware

2.6.1 The Purpose of this Section

The aims of this section are three-fold. The first aim is to illustrate the code necessary to implement a simple client-server program in ANSAware. A practical example will also provide an insight into the structure of a basic ANSAware application. It will also place some of the concepts discussed so far into context.

The example program consists of a server supporting a single operation which adds two numbers together, and a client which invokes the operation. The example demonstrates the concepts of a server exporting an offer to the trader and a client importing a service. Other concepts demonstrated include binding and invocation, and the role of interface references. This section is split into three areas: the definition of an interface, server code and client code.

2.6.2 IDL File

The IDL file *Cabbage.IDL* declares an interface "*Cabbage*" with one operation, *add*. This operation takes two integers, adds them together and returns the result as an integer. This file will be compiled by the ANSAware IDL compiler.

```
Cabbage : INTERFACE =  
BEGIN  
    Add : OPERATION [NumberA : INTEGER; NumberB : INTEGER]  
    RETURNS [INTEGER];  
END.
```

2.6.3 Server Code

The server file, *CabbageServer.DPL* contains all the necessary code for implementing a server. This file will be pre-processed by the *prepc* compiler. DPL files generally consist of three distinct parts:-

- *DPL Statements* - These are always preceded by an exclamation mark which is located at the left hand side of the screen. All statements entered in this manner are currently pre-processed into C files.
- *Body Procedure* - When the server is executed, the body procedure is run. This procedure performs all initialisation functions such as exporting an interface to the trader.
- *Operation Implementations* - Contains code for implementing the operations defined in the interface.

The code listed in the following three subsections is contained in one file in the order below.

2.6.3.1 Prepc Statements

Each source file must declare the names of all interface types referenced within it.

```
! USE Cabbage
! DECLARE {OurIR} : Cabbage SERVER
```

The *USE* statement specifies that the interface *Cabbage* is to be used by the server. In this example all declarations made in the *Cabbage.IDL* file will be accessible to the server code.

Interface reference variables must be related to the interface types to which they are bound. The *DECLARE* statement specifies that *OurIR* will be declared as an interface reference in the server and will relate to server operations on the *Cabbage* interface.

These explicit declarations contribute to the additional type checking necessary to make distributed applications more robust.

2.6.3.2 The Body Procedure

```
void body(argc, argv)
int argc;
char **argv;
{
    ansa_Char Properties[1024];
    ansa_InterfaceRef OurIR;

    system_init_properties(Properties, 1024, argc, argv);
    ! {OurIR} :: Cabbage$Create(16)
    ! {} <- traderRef$Export(Cabbage,"/", Properties, OurIR)
}
```

The body function in this example declares two *ansa* variables⁵, an interface reference and a properties variable (see section 2.5.1). The properties variable is initialised to a default value by the *ansa* call *system_init_properties*.

The *create* operation creates an instance of the *Cabbage* interface and returns a pointer to the interface in the interface reference *OurIR*. The maximum number of invocations that may be processed concurrently is limited to sixteen by the parameter to the *create* function.

Finally, the *Cabbage* interface is exported to the trader using the *export* operation. The

⁵ANSAware variables are prefixed by *ansa_* to avoid type clashes with other software packages such as X-Windows.

parameter "/" indicates that the interface is located directly below root of the type space.

2.6.3.3 Implementing Interface Operations

This source code for implementing the *Add* operation is indicated below and is fairly self-explanatory:-

```
int Cabbage_Add(Attr, NumberA, NumberB, Result)
ansa_InterfaceAttr *Attr;
ansa_Integer      NumberA,
                  NumberB,
                  *Result;
{
    *Result = NumberA + NumberB;
    return SuccessfulInvocation;
}
```

Several interesting differences exist between the ANSAware implementation above and a standard C procedures. Firstly, the operation name *Add* is prefixed by the interface name. The header of the operation differs from its definition in two other respects. A parameter *attr* has been added to the start of the parameter declarations. This is necessary for all procedures implementing interface operations. Return variables defined in the IDL file (i.e. *Result*) are also listed after the parameters to the operation.

The return statement indicates that the invocation has been successful. Other return types are defined by ANSAware. Further information may be found in [APM,93].

2.6.4 Client Code

The client code has many similarities in structure to the server code. *USE* and *DECLARE* statements are necessary. The *DECLARE* statement in this case lists two interface references which are to be used with the *Cabbage* interface by the client. Each client also has a body procedure which is executed when the client is run. This declares two interface references, *TheirIR* and *ServerIR* referred to in the *DECLARE* statement.

```
! USE Cabbage
! DECLARE {TheirIR, ServerIR} : Cabbage CLIENT
void body(argc, argv)
int argc;
char **argv;
{
    ansaInterfaceRef TheirIR,
                    ServerIR;
    ansa_Integer      Result;
!   {TheirIR} <- traderRef$Import("Cabbage", "/", "")
    ifref_copyRef(&ServerIR, &TheirIR);
!   ServerIR$Add(5,7)
    printf("Result = %d\n", Result);
!   ServerIR$Discard
    ifref_freeRef(&ServerIR);
}
```

An integer variable *Result* is declared to hold the result of the *Add* operation. The *Cabbage* interface is imported from the trader using the *import* statement and assigns it to the interface reference *TheirIR*. This interface reference contains pointers into the buffers used for marshalling and unmarshalling which may change over invocations. The *ifref_copyRef*

operation copies the interface reference *TheirIR* into the interface reference *ServerIR*. The copy operation copies all data including the data in the marshalling and unmarshalling buffers into the structure pointed to by *ServerIR* preserving *TheirIR*.

The imported interface reference is then used to invoke the *Add* operation with the parameters '5' and '7'. Once *Add* has been invoked, the discard operation is used to inform the nucleus that the interface reference *ServerIR* is no longer required.

2.7 Summary

This chapter discussed the ANSA architecture, and ANSAware, a partial implementation of the ANSA architecture. The material in this chapter has been presented using concepts detailed in the ODP integrative standard. These concepts include the computational model (which specifies trading, invocations and an object model), the engineering model (which supports the computational model), and ODP functions. The ANSA architecture fits naturally into ODP with a well defined object model at the computational level and support for computational abstractions at the engineering level.

In addition, sample ANSAware code has been used to illustrate the practical application of the ANSA architecture.

Chapter 3 - DCE

3.1 Introduction

OSF DCE is a platform supporting distributed computing over a heterogeneous environment of computers, networks and operating systems. The main goal of DCE is *interoperability*. This is achieved by providing a uniform set of services based, where possible, on proven and existing standards and methods.

Unlike many other distributed platforms, DCE was designed to be of commercial strength and has been endorsed by many consultants, users and vendors. The component technologies of DCE were selected from industry submissions using the OSF request for technology (RFT) process⁶. A RFT specifies a technology, related constraints and conformance points (such as interfaces, and data types). Industry is then invited to submit solutions to OSF for evaluation and the most appropriate solution is selected. The vendor-neutral position of OSF ensures that technologies are evaluated only on their individual merits. The submissions and selected solutions for the DCE services are listed below:-

Technology	Submissions (selected technology in italics)
Threads	<i>Concert Multithread Architecture - CMA (Digital)</i>
Remote Procedure Call	<i>Network Computing System (Digital & Hewlett-Packard)</i> Netwise RPCTool (Sun Microsystems) Open Network Computing (ONC) RPC (Sun Microsystems)
Cell naming	<i>Distributed Naming Service - DECdns (Digital)</i> Network Directory Service (Hewlett-Packard)
Global naming	<i>DIR-X X.500 Service (Siemens)</i> DS-520 X.500 Service (Retix)
Time	<i>Distributed Time Synchronisation Service - DECdts (Digital)</i> Network Time Protocol (Transarc)
Security	<i>Project Athena's Kerberos (MIT)</i> UniDesk/Config (Unicare) Authentication/Authorisation (Hewlett-Packard)
Files	<i>Andrew File System version 4 (Transarc)</i> Sun Network File System version 3 (subsequently withdrawn)

⁶The OSF announced a restructuring process known as "NewOrg" in March of this year. The RFT process remains to allow OSF to stimulate new technology, in conjunction with a new Pre-Structured Technology (PST) process. The PST approach differs from the RFT in that it is industry-initiated, industry-sponsored and highly project-driven approach promoting increased industry cooperation.

Personal Computer Integration/ Diskless Support	<i>PC-NFS (Sun Microsystems)</i> <i>LAN Manager (Microsoft & Hewlett-Packard)</i> PCILIB, PC-Interface, Merge (Locus)
--	---

The impact of DCE on industry is significant and the environment is now an emerging de-facto standard for distributed computing.

ODP has had little influence on the DCE architecture which takes a service-oriented approach towards distribution. Thus DCE consists of a set of integrated services which have no direct correlation to many of the ODP abstractions. The description of DCE in this section will be oriented towards the services provided and their respective implementations.

3.2 An Overview of DCE

DCE adopts a client-server approach and supports encapsulation at the server-level. DCE provides a programmer with a set of function calls in C for performing distributed actions, such as creating bindings and invoking server operations. Thus, DCE code is mainly standard C with no abstraction for distribution. The only exception is an Interface Definition Language (IDL) which specifies the interfaces of server objects. Particular features of interfaces may be modified using an *attribute configuration file* (ACF).

The DCE architecture provides a set of eight services which are split into *fundamental services*, providing programmers with tools with which they can write distributed applications, and *data sharing services*, which are built on the fundamental services and provide end-user support for distribution. The general DCE architecture, illustrated below, is built over a supporting operating system.

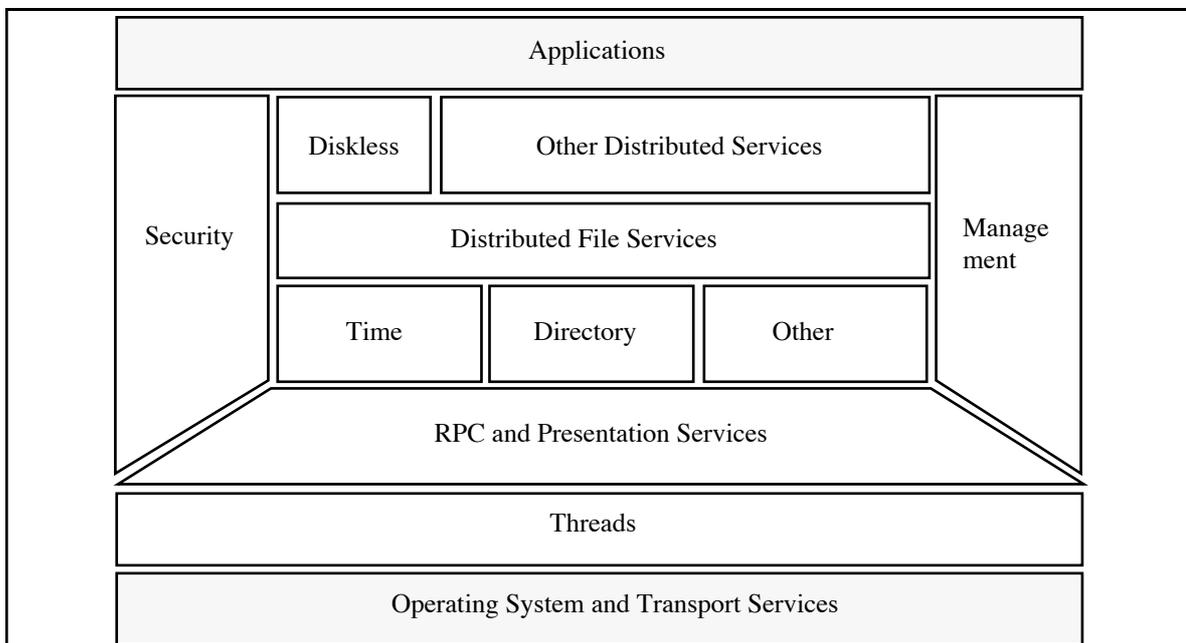


Figure 3.1 - the DCE architecture

The following sections will look at each of these services in turn⁷. We will start with the

⁷Note that the management service consists of a set of configuration tools and C routines. Each of these are integrated with a particular DCE service. Management tools applicable to a particular

services closest to the operating system and progress to the higher-level services.

3.3 The Fundamental Services

3.3.1 The Threads Service

A *threads service* is provided at the lowest level and is used by most of the other DCE services to provide a concurrency mechanism. A thread is a sequential flow of control within an address space. Several threads may share the same address space.

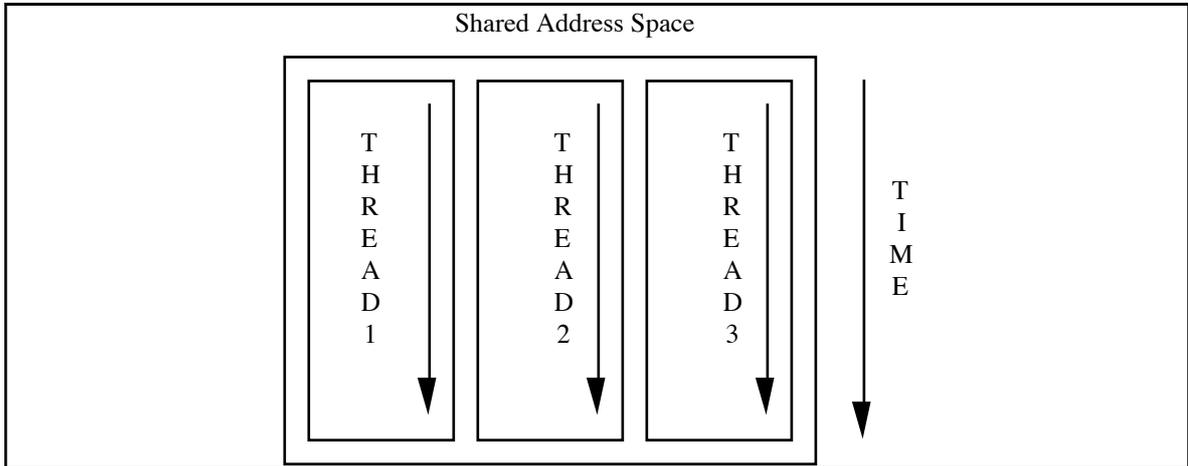


Figure 3.2 - threads share an address space

The concept of a shared address space can be an advantage if used carefully. The threads illustrated may use shared variables to share state and communicate efficiently. Control over the access to shared information is necessary for reasons of consistency and is provided by *mutual exclusion* and *condition variables*.

Threads allow multiple client-server configurations. For example, a client may invoke several operations on a particular server and/or a range of servers. Each invocation will generate a new thread in the specified server. Conversely, multi-threaded clients may be created which invoke multiple servers simultaneously.

The threads service provides specific calls for the creation and termination of threads in DCE. Facilities such as the join, mutex and condition variables are also available to control synchronisation policies to critical areas of code and data (such as shared variables). It is also possible to make specific calls for setting thread states (in DCE, a thread may be running, ready, blocked or terminated).

These interactions with the threads service are through a threads application programmers interface (API). For example, the C routine *pthread_create()* may be used to create a thread and *pthread_detach()* requests a running thread to terminate. An exception handling mechanism is supported in order to return error values to calling routines. Programmers generally do not concern themselves with threads programming as threads are usually invoked by calls to other services such as RPC or the directory service.

When programming threads, care must be taken to avoid situations such as deadlocks, race conditions and priority inversion. Possible deadlocks will occur between threads if, for example, two threads are waiting to use a resource that the other one holds. Race conditions must also be avoided, where two threads have inconsistent views of the same variable which leads to unpredictable results. Priority inversion occurs when three threads interact in such a way that the highest priority thread is blocked and lower priority threads may execute.

service are detailed in the relevant section.

3.3.2 Threads Implementation

DCE threads is based on the DEC Concert Multithreaded Architecture (DEC CMA) which, in turn, is based on the POSIX 1003.4a Draft 4 pthreads standard [IEEE,94]. The threads package is a user level (non-kernel) implementation. In a UNIX environment, the shared address space is a UNIX process. Processes may be single-threaded or multi-threaded. Threads occupying the same process may share variables with each other.

Several problems exist with multi-threaded applications. In particular, the code generated for multi-threaded applications must be:-

re-entrant

Compilers must generate reentrant code so that routines may be executed concurrently and successfully.

thread-safe

All libraries must be thread-safe. By “thread safe”, we mean that certain services may only be accessed by one thread at a time (such as when two threads attempt to write to the same file) or that the effect of a particular thread operation must be controlled. Jacket routines are provided by the threads package which surround many UNIX calls to make the UNIX library thread-safe. When a UNIX call with an associated jacket routine is invoked, control is first passed to the jacket routine. The routine typically passes control to the UNIX call only if no other threads are currently using it. For example, consider a thread which issues a call which causes the whole process to block. If jacket routines were not provided then all other threads running in the same process would also be blocked without their consent, with possibly disastrous consequences. Such asynchronous blocking is unacceptable.

Scheduling must be controlled at the process level and the thread level. Scheduling is carried out at the process level by the host operating system and at the thread level by the threads package. This is an example of an uncoordinated split-level scheduling technique. The three scheduling policies offered are *prioritised FIFO*, *prioritised round robin* and the *default timesliced* policy. Prioritised FIFO queues involve multiple priority FIFO queues of threads, where the highest priority thread runs until it blocks, then the rest of the same priority followed by the lower priority queues. The round robin scheduling policy allows the highest priority thread to run. Any threads of the same priority are timesliced. Finally, the default policy timeslices threads with no account given to their priority. This ensures that priority inversion cannot occur. Thus scheduling is at the individual thread level, each thread is on a particular queue and has its own scheduling policy. Threads using the default policy are scheduled first, followed by threads scheduled by the other two policies.

3.3.3 The Remote Procedure Call Service

The *remote procedure call* is the method of interprocess communication (IPC) provided by DCE. This service is layered on top of the threads service and essentially provides access and location transparencies. Servers may export an interface to the directory service and clients may import interfaces. This establishes a binding. When a client invokes an operation on an interface, the RPC mechanism transmits the invocation request from the client to the server. The invocation is implemented at the server as a local procedure call. The server then transmits any results back to the client. A consistent, uniform view of data types at the RPC level is provided by the *presentation services*. These abstract over machine specifics such as data types and byte ordering. Thus, RPC in conjunction with the presentation services allows a programmer to make invocations on objects without concern for whether the target object is located on the same machine or on a remote machine. Furthermore, RPC's may be made transparently over heterogeneous machine architectures (i.e. hardware and operating systems).

Interface definitions for objects are specified using an IDL. The IDL is a declarative language which declares operations as procedure headers with ordered parameters to the procedure and possibly a return type. An IDL interface consists of a *header* and a *body* (see section 3.5). Attributes specified in the IDL include:-

Interface Header Attributes

Specify the *universal unique identifier* (UUID), which uniquely identifies an interface globally in a distributed system. Other attributes include the interface version number for version control, default treatment for pointer types, well known endpoints (see next section) and stub generation options. Interface header attributes affect all interface data types and operations.

Data Type Attributes

Includes array attributes for specifying array bounds and sizes and attributes for classifying pointers. Further attributes exist for structures and unions. Data type attributes are declared in the interface body.

Procedure Attributes

Procedure attributes affect individual operations and are also specified in the interface body. Procedure execution attributes specify whether procedure invocations are broadcast to all hosts on the network providing a particular service or a specific server. A procedure may be defined as having maybe semantics or idempotent. Several procedure parameter attributes exist. A typical example of such attributes are the in and out attributes which specify whether a parameter is a reference or value parameter. The third class of procedure attributes are procedure result parameters which define whether parameters are string or pointer parameters. This class of attributes may also maintain a pointer to a state on a specific server for reference in future invocations.

DCE RPC allows control over some aspects of RPC on the client side without affecting the server. The opposite is also true. This is achieved by specifying additional attributes in an *attribute configuration file* (ACF). An ACF separate to the IDL file is necessary to avoid forcing the attributes on all clients and servers. Thus attributes specified in the ACF modify the way in which stubs are created without modifying the way in which they interact across the network. Each client and server may have an associated ACF. An ACF may specify the following attributes:-

- *Binding Methods* - binding may be automatic, implicit or explicit (see implementation section).
- *Exception Handling* - allows exception status codes to be passed out of procedures as named parameters instead of generating exceptions.
- *Procedure Exclusion* - excludes code for unused procedures in an interface from the client stub code.
- *Marshalling* - controls whether code for marshalling/unmarshalling is generated in the stub code or auxiliary files.
- *Miscellaneous* - includes memory and data type management attributes.

A server may export several interfaces with different operations and attributes. Furthermore, DCE RPC introduces the concept of an RPC object by allowing the association of an *object UUID* with a particular RPC object. Example RPC objects include a server, a database entry, a device, a directory or a database. Object UUID's are useful in cases where a client may wish to distinguish between servers with identical interfaces, or servers offering alternative implementations of the same interface. If servers offer distinct computing resources such as access to a particular printer, RPC objects are associated with

each printer. A server may access the correct printer using object UUID's.

DCE RPC is closely linked with the security and directory services. The reader is directed to sections 3.3.5 and 3.3.9 for further information.

3.3.4 Remote Procedure Call Implementation

Access to the RPC facilities is provided by *RPC runtime* which manages communications and provides a library of supporting C routines (the RPC API). The services provided by the runtime can be split into five basic categories:-

- *Communications Routines* - which transmit, receive and establish bindings.
- *Directory Service Operations* - for accessing the name service databases.
- *Endpoint Operations* - for manipulating addressing information at the local cell.
- *Security Operations* - for authentication/authorisation purposes.
- *Miscellaneous Operations* - including management (starting/stopping servers) and UUID manipulation.

DCE provides several tools for the manipulation of RPC components. Two examples are included for illustration purposes (the reader is directed to the appropriate documentation for a complete list of available tools [OSF,92a]). A UUID generator *uuidgen* may be used to generate a skeleton IDL file. This contains an interface UUID and an interface version number. The skeleton IDL file can then be filled in with constants, types and operations as appropriate. An RPC control program *rpccp* allows a programmer or user to interactively modify RPC information registered in the name service, e.g. as bindings and endpoint mappings.

The DCE RPC provides an IDL compiler which compiles interfaces into header files for inclusion into client and server code, and C client and server stub routines. The stub routines perform marshalling and unmarshalling and conversions between data representations (i.e. the presentation services). The client stub routines will also locate an appropriate server if the automatic binding method is selected in the ACF (binding methods are more fully described later in this section).

The mechanism of a remote procedure call is illustrated in figure 3.3. This diagram indicates how the different parts of the RPC service link together.

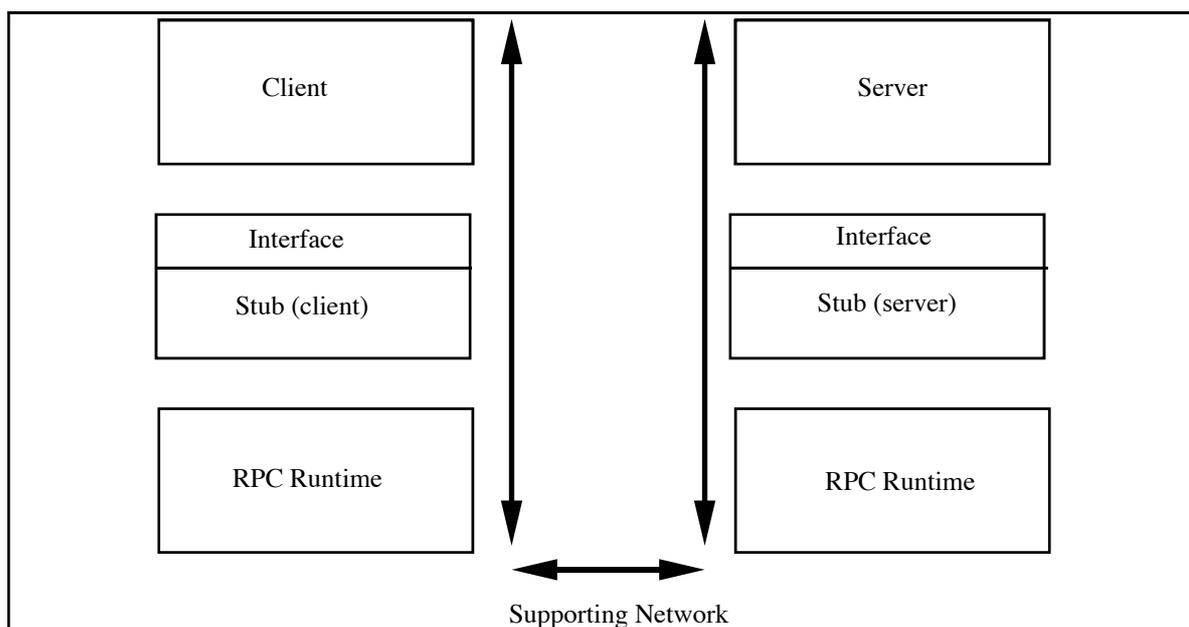


Figure 3.3 - the mechanism of DCE RPC

After importing an interface from the directory service and establishing a binding, the client code issues an RPC. Any arguments in the RPC are passed to the client stub which marshalls the arguments into the relevant format and sends the call to the RPC runtime. DCE supports UDP/IP as a connectionless protocol (datagrams) and TCP/IP as a connection-oriented protocol. RPC runtime transmits the call over the network to the appropriate server stub which uses its own copy of the RPC runtime to unmarshall the arguments and convert them into the machine's local format before passing them to the relevant local procedure. Transmissions are encoded in Network Data Representation Transfer Syntax (NDR). Once the procedure has executed, the return arguments are passed back to the calling procedure in a similar manner to how they were transmitted.

The three methods of locating a server (automatic, implicit and explicit) differ substantially and require varying levels of engineering support. The automatic method means that the client stub locates the server and binds to it transparently. This requires the most engineering support. With this option the user has no control over which server is chosen and is only aware that a server with the correct functionality has been selected. If a server fails, the client automatically attempts to bind to another server. Less engineering support is required for the implicit method. This requires the programmer to locate the binding information for a server and store this in a global binding handle in the client stub. This information is passed to RPC runtime by the stub. Thus, a client must locate the required server manually. The explicit method requires the least engineering support. Binding information is located by the client application code and is passed explicitly as the first parameter in invocations. The explicit method differs from the automatic and implicit methods in that a binding is on a *per-invocation* basis, i.e. an explicit server may be chosen for each invocation.

DCE RPC includes presentation services. These perform all the necessary conversions between data formats, for example between a machine using ASCII and one using EBCDIC. The presentation services use a 'receiver makes it right' scheme. Thus, the receiver is responsible for converting foreign data into a format that it can use. The RPC mechanism tags types with a description to allow the receiver to perform any necessary conversions. This method differs from canonical forms of data representation such as ASN.1 BER [Neufeld,92]. A programmer may optionally select ASN.1 BER if preferred.

DCE RPC supports a two-level mechanism for locating a server. Servers register their *endpoints* (a pointer to a specific instance of a server, such as a UNIX process ID) with the RPC daemon *rpcd*. The RPC daemon is also known as an "endpoint mapper". The *rpcd* runs as a process on each machine in a DCE network. Thus, *rpcd* provides a local endpoint mapping service, looking up endpoints for RPC clients and maintaining the endpoint database. Figure 3.4 illustrates this process:-

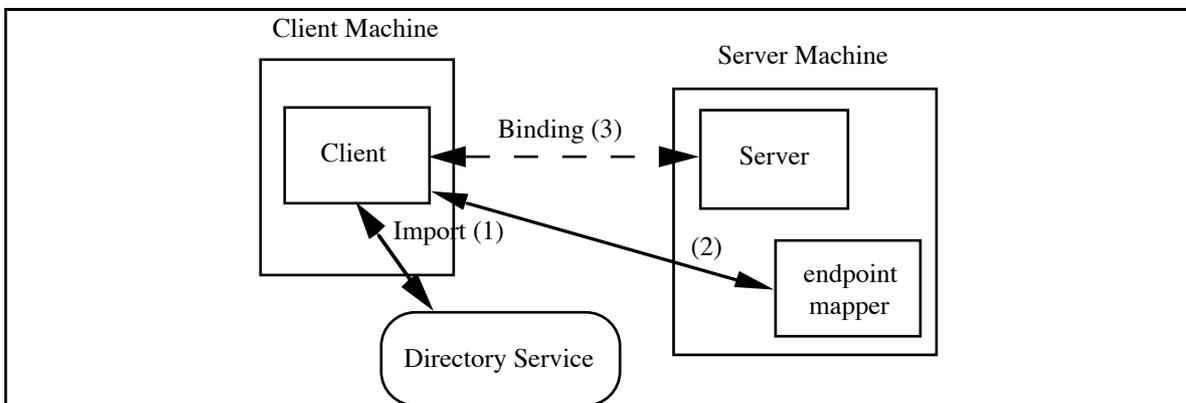


Figure 3.4 - locating a server in DCE

Every server interface is registered separately with rpcd, as is every RPC object offered with the interface. When binding information is retrieved from the directory service for the first time, this is only a *partial binding* (1). This indicates the address of the machine hosting the server process, i.e. where the rpcd for the machine hosting the server can be located. A partial binding has no endpoint information and therefore rpcd must locate a suitable offer in the map database (2) and fill out the partial binding information to create a *full binding* (3). This is returned to the client runtime process which then makes the call. Subsequent calls to this server do not need to use rpcd as a full binding to the server process is now available. The endpoints described here are *dynamic* endpoints and are assigned at runtime. It is also possible to assign *well known* endpoints which are a preassigned stable address that a server can use every time that it runs.

RPC runtime is linked to the security runtime (described in the section 3.3.9). The directory service is accessed through the Name Service Interface (NSI) which provides facilities for the location of registered servers.

3.3.5 The Directory Service

A *directory service* is provided to allow servers to advertise services and clients to locate services. The directory service is a highly object-oriented, replicated distributed database where each entry in the service is considered an object. The directory service is based around the X/Open Object Management API (XOM API) [X/Open,94]. This provides a facility for object manipulation. Typical operations include of creating, modifying and deleting objects. The directory service hides the distributed nature of a system by providing location services in a dynamic environment and is also used by other DCE services, applications and users.

The directory service is split into a *cell directory service*, which deals with machines in the local *cell*, and a *global directory service*, which seamlessly links cells together (figure 3.12). A cell is a local collection of machines and peripherals and usually corresponds to a geographically located set of machines, e.g. a particular department or working group. A cell, however is a logical abstraction and no geographic restrictions exist on cell membership. An *entry* in the CDS points to an *object* and is referenced by a *name*. A collection of entries makes up the *namespace* of the directory service.

Since the machines in a cell are generally co-located, queries to the local cell directory service are generally of higher performance than queries to the global directory service. Thus, each cell has its own directory service and is linked to other cells via the global directory service which provides a *global naming scheme*. This allows any machine in a distributed system to be identified by a global and unique name.

A programming interface, the X/Open Directory Service API (XDS API) [X/Open,94], is provided to allow operations such as lookup, deletion, modification, creation and searching of entries. The XDS API uses the XOM API in order to access and manipulate information. The directory service provides a global uniform naming scheme and, as such, a user cannot distinguish between queries to the cell directory service and queries to the global directory service. The XDS API examines the names of the target objects and passes the query to the CDS or the GDS. If a query to a specific cell namespace fails, the query is automatically propagated to the global service. Queries may be yellow page queries such as 'locate me all the colour printing facilities locally available' or white page such as 'where is this particular server?'.
A typical namespace entry is illustrated below:-

A typical namespace entry is illustrated below:-

```
./.../C=UK/O=LANCUNI/OU=COMP/home/columbine/pha/docs
```

The entry consists of a GDS part (illustrated in capitals) which specifies the Computing Department at Lancaster University in the UK (“/...” specifies the global root). The remainder of the entry (in lower case) is a local cell name which specifies the location of a 'docs' object within the cell.

The directory service provides several administration tools, *cdscp* for CDS and *gdssysadm*, *gdsditadm*, *gdscacheadm* for GDS. A motif-based CDS browser allows a user to inspect the CDS namespace.

3.3.6 Directory Service Implementation

3.3.6.1 The Cell Directory Service

The architecture of a simple CDS lookup is illustrated below:-

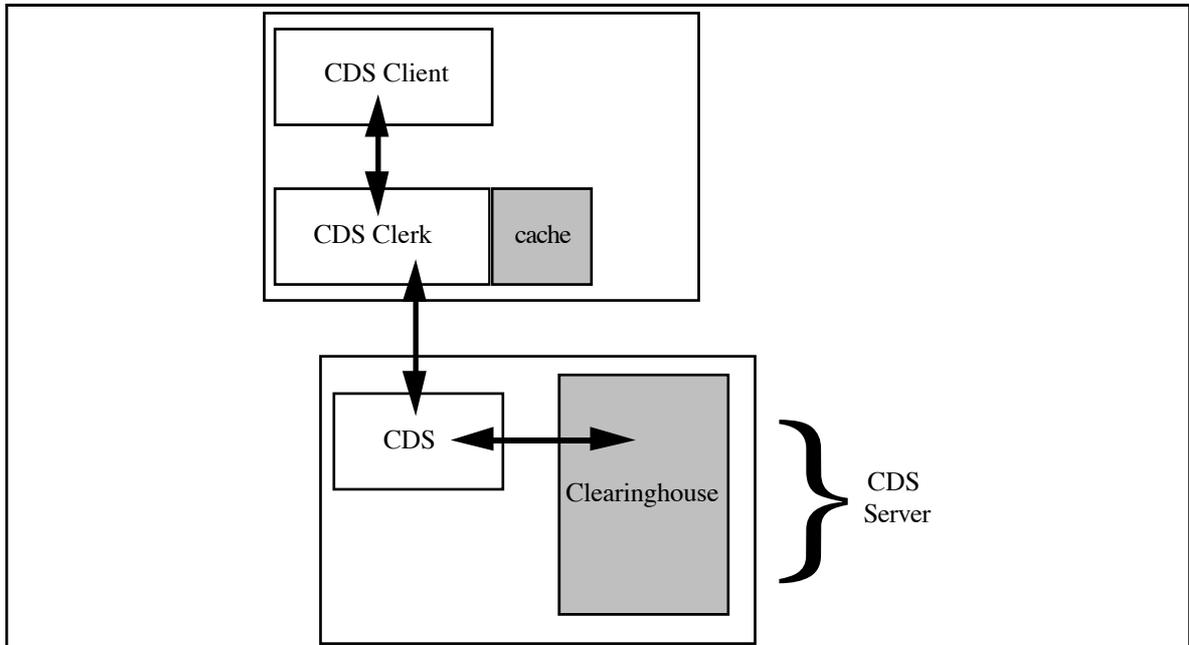


Figure 3.5 - the CDS mechanism

The cell directory service consists of a *clearinghouse* (replicated partitioned distributed database) of entries and a CDS server with at least one CDS per cell. A *CDS clerk* is created for every client application using the CDS. The clerk provides an intermediary between the CDS client and server. Facilities provided by the clerk include the caching of database information to minimise on cell queries across nodes.

Multiple CDS clerks on a single node share the same cache. Thus when an enquiry is made to the CDS via the clerk, the clerk first checks the cache. An enquiry is only forwarded to the CDS server if the information is unavailable in the cache. The CDS server returns the information to the application via the clerk. The clerk will then cache the information. An obvious possibility for cache inconsistency exists. CDS assumes the cache information to be correct and makes an enquiry based on the cache information available. If this proves to be invalid, a query is then made to the CDS server and the appropriate cache entry is updated.

3.3.6.2 The Global Directory Service

GDS allows interaction between cells by providing a worldwide directory service based on the CCITT X.500/ISO 9594 standard [ISO,94]. Support is also provided for the Internet Domain Name System (DNS) [IAB,94], another global directory standard. GDS, like CDS is a replicated distributed database known as the *Directory Information Base* (DIB).

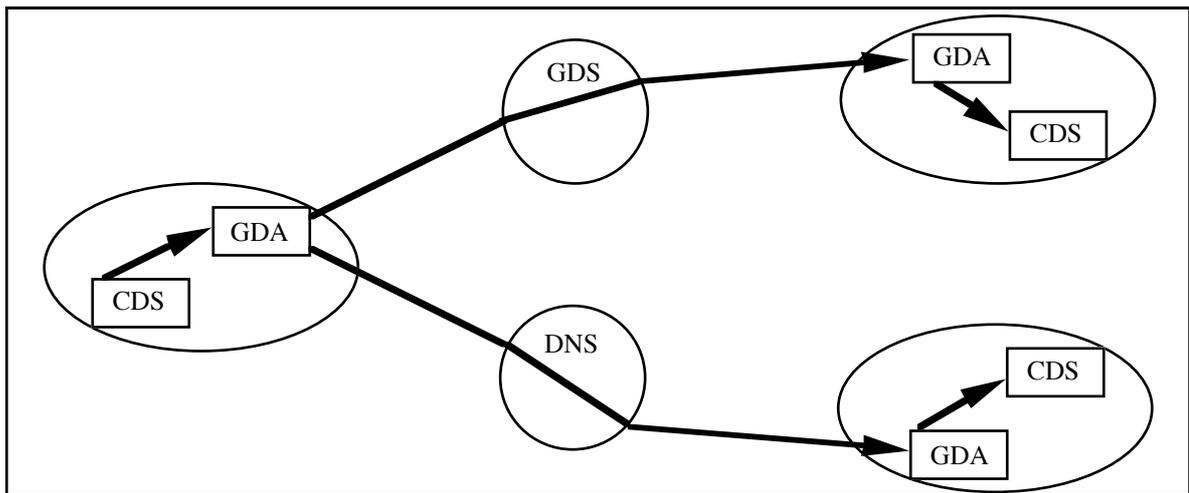


Figure 3.6 - the Global Directory Agent

The CDS and GDS are linked by a *Global Directory Agent (GDA)*, a naming gateway between the CDS and GDS which allows inter-cell lookups. Whenever the CDS is passed a global name, the query is passed to the GDA which forwards requests to either GDS or DNS depending on the format of the name. The response to the query is then passed back to the CDS via the GDA which replies to the client application. The GDA is a separate process to the CDS and GDS servers and may or may not run on the same machine.

3.3.7 The Time Service

A common problem in distributed systems is that the different nodes of a system have separate clocks which inevitably means that each system has a different notion of time. Even though all clocks in a distributed system may be synchronised at system startup, there is always an element of *skew* between different clocks. DCE provides a *distributed time service* which gives a system wide notion of time synchronised to *coordinated universal time (UTC)*. This is an international time standard that is beginning to replace Greenwich Mean Time (GMT).

Interacting with the time service involves manipulating *timestamps* which DTS uses internally to represent time with an inaccuracy factor (this is for synchronisation purposes and will be discussed in the implementation in section 3.3.8). These are opaque binary timestamps and cannot be read or disassembled manually. A programming interface, DTS API, allows a user to retrieve timestamp information and convert the binary timestamps to ASCII text strings which may then be displayed and manipulated. The DTS API also allows conversion between different timestamp structures, comparison of timestamps, calculation of time zone information and conversion between local time and UTC. Facilities are also provided to allow access to the local system clock if preferred. An ASCII timestamp is illustrated below for reference purposes:-

`1993-11-01-11:25:31.65357-03:001101.07370`

Timestamps represent a particular point on a time scale (absolute time). The absolute time structures supported by the time service are as follows:-

- *UTC* - this is based on 100-nanosecond time units from a base date of 15 October 1582.
- *tm* - a GMT structure based on 1-second time units from 1 January 1900.
- *timespec* - specifies the number of seconds and nanoseconds which have passed since 1 January 1970.

DTS also provides relative time which is an interval that is added/subtracted from an

absolute time. An example of this is the time difference factor (TDF) component of the timestamp which is used in calculating time zones. A *time provider interface* DTS TPI allows application programmers to synchronise with an external UTC time source. The mechanics of external time providers will be discussed in the following section.

3.3.8 Time Service Implementation

The distributed time service is based on Digital's Distributed Time Service (DEC dts) and is interoperable with the Internet Network Time Protocol (NTP) [IAB,94]. The time service is a client-server system and uses RPC and the directory service to locate time servers.

Additional complexities exist in maintaining an accurate representation of time in a distributed system. Minor discrepancies will exist between the clocks in different machines as they drift apart over time. Communications between machines (e.g. for clock synchronisation) introduce indeterminacy. A new mechanism for time representation is required which includes not only an expression of the current time but an inaccuracy factor. Time is represented as an interval within which the correct UTC time is known to lie. The 'correct' value of time is taken to be the midpoint of the time interval figure (see figure 3.7).

DTS time is represented internally using a 128-bit timestamp.

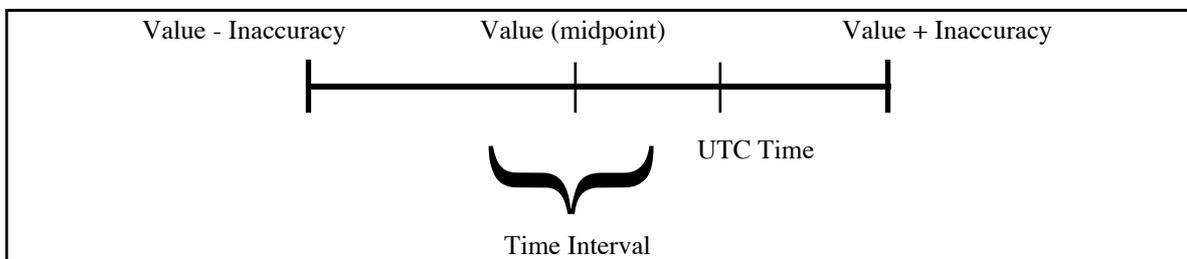


Figure 3.7 - time value and inaccuracy

The architecture of DTS is based on *clerks* (clients) and servers which synchronise with each other periodically to minimise skew (figure 3.8). Four different types of time server exist:-

- *Local Servers* - synchronise with similar servers on the same LAN.
- *Global Servers* - provide synchronisation with an extended LAN or WAN.
- *Courier Servers* - are local servers which synchronise with global servers.
- *Time Provider Servers* - provide time from an external UTC time provider (a satellite link or radio link to a reliable external time source for example).

It is not necessary to have an external time provider in a system. However, it possible for clocks to collectively drift away from UTC without an external source of time. A cell generally has several time servers (three is a typical number with one server designated as a global server).

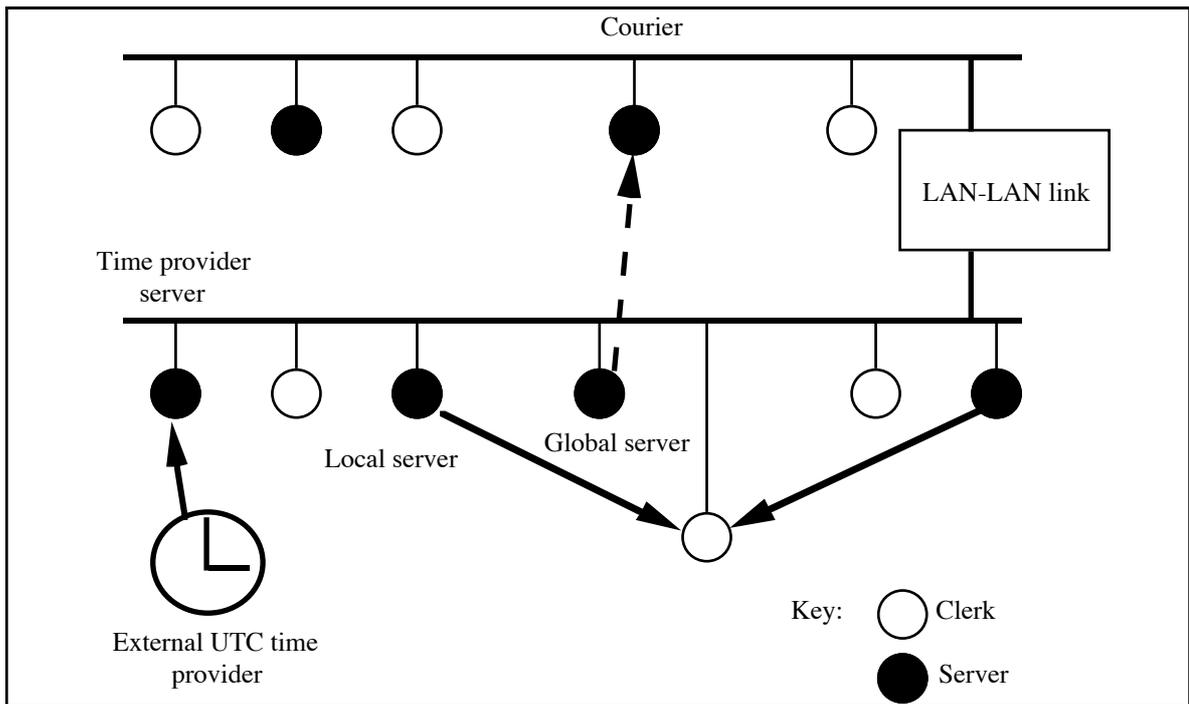


Figure 3.8 - the time service architecture

Three types of synchronisation occur at the implementation level:-

Clerk Synchronisation

It is always possible for DTS servers to become inaccurate or to fail completely. A DTS clerk that requires the time therefore requests that servers send their combined clock and inaccuracy values and a new system time is computed using the midpoint of the intersection of the returned times. Clerks adjust the system clocks of their machines accordingly. This reduces the skew between systems as the more synchronisations that are performed result in smaller intervals than that supplied by a single server. If a server requests a time that does not intersect with a large proportion of the other times, that server is deemed faulty and the DTS administrator is informed.

System Synchronisation

When a clerk adjusts the time of the system clock to synchronise with the other clocks in the system, this is done by incrementally modifying the length of a clock tick on the system to take slightly more or less time depending on whether the system is running too fast or too slow. The adjustment is one unit per hundred, i.e. it will take 100 seconds to repair an error of 1 second.

Server Clock Synchronisation

Before this synchronisation is performed, the DTS first checks to see if an external time provider exists for the server system. If one is available, the server synchronises with this imported time. This is then propagated through the network. If no external time provider is available, synchronisation is performed with the other servers. Synchronisation is with a peer server on the local LAN. Skew between servers on a LAN is usually less than 200ms. While this occurs, a courier will be synchronising with a global time server outside the local LAN. This process results in a global notion of synchronised time. Servers with the greatest accuracy have the most influence on synchronisation.

3.3.9 The Security Service

A distributed system by its very nature provides ample opportunity for security breaches. For example, an unscrupulous individual or enterprise may eavesdrop on an unprotected communications line and obtain sensitive or confidential information. Possibilities also exist for the substitution of genuine data with forged information and impersonation. Consider an insecure distributed system where an accountant sends sensitive financial information to a printer. It is possible to impersonate a printer with a process which collects information destined for the printer and then forwards it to the correct destination afterwards. The accountant would not be aware that there had been a security breach as the information would still be printed correctly. It is therefore obvious that some form of protection is required in order to make distributed systems a viable possibility for enterprises. DCE provides a *security service* with four basic functions:

- *authentication* - the security service authenticates the identities of *principles* (entities in the security service such as users and applications) so that the system knows that who they claim to be is correct.
- *authorisation* - once the identity of a principal has been established, it is necessary to check that they are authorised to use the service that they are requesting or the action that they are trying to perform (possibly a modification to a file).
- *verification* - the security service verifies the integrity of data to check for data corruption or unauthorised modification.
- *data privacy* - sensitive information may be encrypted in order to render it unusable to all but the authorised users of such data.

Each of these facilities are chosen selectively, allowing applications to implement the desired level of security.

The security service consists of three major services. These are support the authentication and authorisation functions:-

- *The Registry Service* - which manages a database of users and groups with their accounts and passwords.
- *The Authentication Service* - providing the authentication facility.
- *The Privilege Service* - which certifies the credentials of principles and grants access privileges.

Users of a DCE system log into the DCE machines. Users are authenticated against the list of users and passwords stored in the registry service. A current topic of discussion at OSF is the possibility of allowing a user to log on to a DCE network and have access to all machines on the network as an authenticated principal, rather than having to log on to each individual machine or having to be authenticated every time they wish to transfer files to another user.

A security API is provided to allow applications programmers to implement security services. Each application must provide an *access control list* (ACL) manager in order to interface with the security service. ACL's are used for authorisation purposes. A utility, *acl_edit* is provided to allow users to create, delete, modify and list the permissions of principals and groups.

3.3.10 Security Service Implementation

The DCE security service is based on Kerberos Authentication from MIT, POSIX 1003.6 Access Control Lists (ACL's) [IEEE,94] and the Data Encryption Standard (DES).

Authentication is supported by private key technology which involves the exchanging of secret messages between principals. Authorisation is based around a superset of ACL's. An

ACL is associated with a resource, containing a list of principal names and their permissions (the types of operations they are allowed to perform). When a service is requested, the requester must be listed in the appropriate ACL and it must have the appropriate permission for the action that it wishes to perform. Permissions may be user defined and not necessarily the standard UNIX read, write and execute permissions. An ACL entry takes the format **Type:Identity:Permissions:[Class]** where *type* identifies the role of the entry, *identity* identifies the principal (i.e. a username), *permissions* identifies the access rights of the user and *class* optionally identifies a mask that can be used against the permissions of the principal.

These concepts are illustrated below:-

group_mask:-rwx---

user:phil:-rwx---

foreign_user:steve@uk.ac.lancs.comp:-rwxid-

Thus, the user 'phil' has read, write and execute permissions on the object associated with the ACL as does the user 'steve' when the group mask is applied.

Data integrity is verified by using encryption keys to generate cryptographic checksums which prevents the unauthorised tampering of data while it travels across the network. Data privacy is ensured through trusted third party secret key encryption. Both parties have an encryption key and encrypt/decrypt the data before/after transmission.

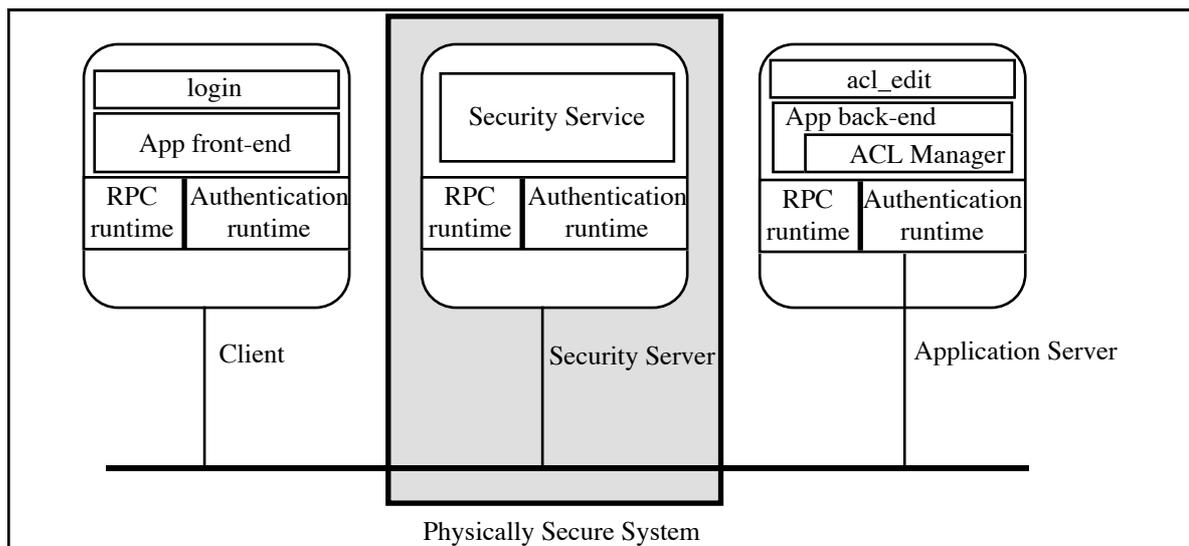


Figure 3.9 - the components of the security system

The components of the security service are illustrated above. Each cell must have a security server which should be physically secure to prevent unauthorised physical access. The security server consists of the security service component (registry service, authentication service and authorisation service) which sits on top of an authenticated RPC service (an authentication runtime process and a RPC runtime process). The security client contains a login facility and the application front-end i.e. the client side of the application which sits on top of authenticated RPC. The application server contains the application back-end i.e. the server processes which contain an ACL manager and use authenticated RPC.

The authentication and authorisation process can be best illustrated by example:-

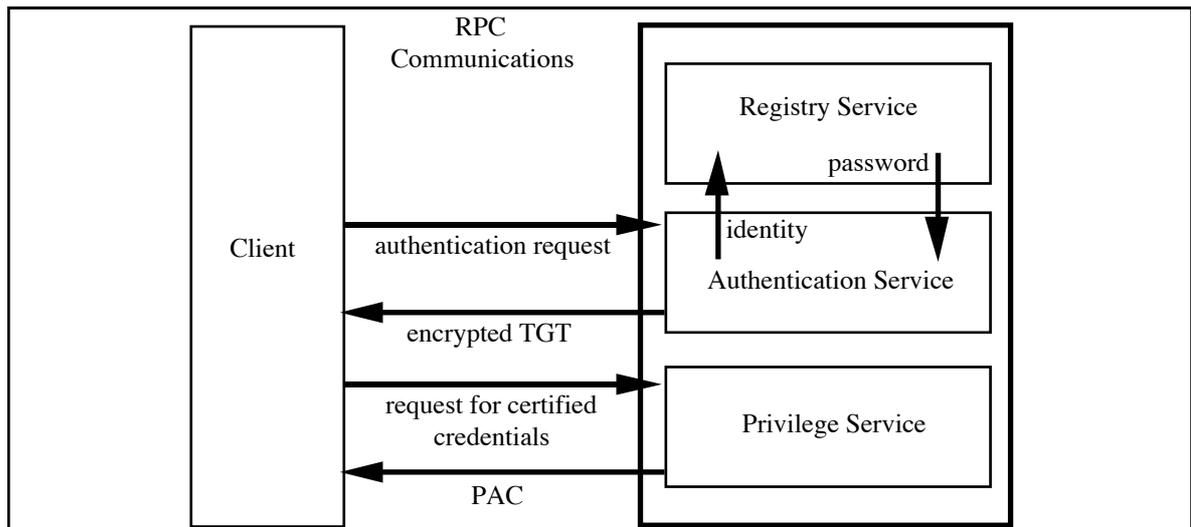


Figure 3.10 - authentication and authorisation

A user logs in via the login utility. The username is passed to the authentication service which gets the user's password from the registry service. The authentication returns a *ticket granting ticket* (TGT) to the client process which is encrypted using a secret key derived from the user's password. The client process decrypts the TGT (it derives the same secret key as the authentication service from the password specified by the user) and requests credentials from the security service. The security service verifies that the client has successfully decrypted the TGT and therefore is a trusted principal and issues a *privilege attribute certificate* (PAC). The client process is now in a position to request that servers perform operations. When a client requests an operation in an object, it requests a *ticket* from the privilege service to access the desired server. A ticket is server-specific and has a finite lifetime. Once a ticket has expired, another must be requested. The privilege service returns the ticket and the client requests the desired operation. The server uses the client credentials in the ticket and compares them to the ACL entry for the object. If the client has the required permissions then the operation is executed. This example ignores the encryption of messages using secret keys.

3.4 Data Sharing Services

3.4.1 The File Service

The *distributed file service* provides location-independent file access over local and remote systems eliminating the need to transfer files from one system to another. It provides high performance file access and is closely integrated with the security, directory, time, threads and remote procedure call services. The DFS is a member of the data sharing services and is generally transparent to users who simply perceive a uniform, consistent file service and naming scheme. Users have no notion of the distribution of the underlying file systems.

The DFS manages file system objects (i.e. directories and files) and provides functions for cache management and the manipulation and location of *filesets* (a logical unit in the file system) via a DFS API. Filesets are more fully described in the implementation section.

3.4.2 File Service Implementation

The Distributed File System is based on the Andrew's File System (AFS) [Levy,90] and uses POSIX 1003.1 file system semantics [IEEE,94], POSIX 1003.6 access control [IEEE,94] for security and is interoperable with NFS [Levy,90].

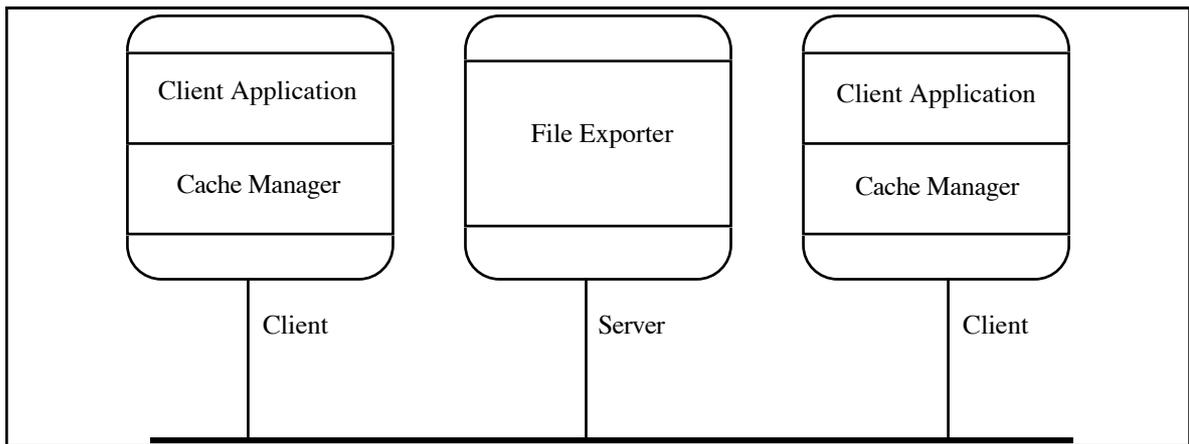


Figure 3.11 - file system architecture

The system is based around a client-server approach providing a replicated distributed database. Each client of the file system manages a cache for reasons of maximising performance and minimising server load and network traffic. Caching provides many benefits but creates the problem of cache consistency e.g. if a file that is cached is written to by another client. The distributed file system solves this by associating a *token* with the cached file. DFS provides controls over which file can be written. To obtain access to a file, it is necessary to obtain a token which gives permission to use the file. Example tokens are read and write tokens. Several read tokens may be issued but only one write token may be issued for a file. If a file is modified, the tokens issued for the file become invalid and a new token must be requested in order to obtain a new copy of the file.

DCE file servers are based on the Episode-based local file system (LFS) although other file systems may be substituted. LFS is a log-based high performance, highly available and secure file system providing fast restarting in the event of failure. A log-based system keeps logs of all transactions in log files. This allows fast restarting of crashed file servers. The server provides location transparency, uniform naming and consistency control. The system makes use of the directory service to provide a uniform naming scheme (figure 3.7). An example of the use of the naming scheme provided by the directory service is the convention of creating a subdirectory from the cell root labelled 'fs' for the file system and another labelled 'sec' for the security system.

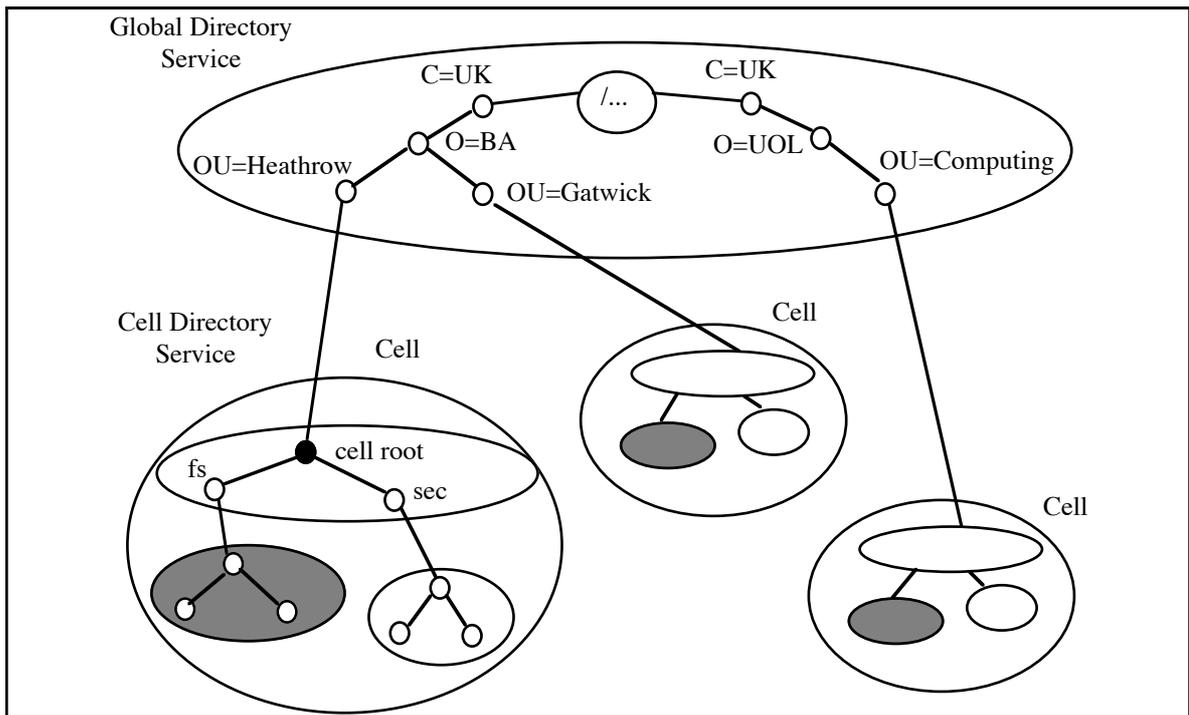


Figure 3.12 - uniform naming scheme of the file system

This allows administration to be performed while the file server is running (backups, file moves). The basic unit of manipulation (replication, copying etc.) is the *fileset*. A single fileset refers to many files, therefore when a particular file is specified, the relevant fileset is located transparently and the correct file is accessed. Filesets are held in *aggregates* which are physically equivalent to a UNIX disk partition. Filesets are logically equivalent to directories, i.e. a hierarchical grouping of files managed as a single unit.

The file service is closely linked with the security service which uses ACL's to confer access rights to files.

3.4.3 The Diskless Support Service

A *diskless support* service is provided to support workstations without any local secondary storage. These are preferred by some enterprises as they are cheaper to buy and provide security advantages (it is impossible to upload software such as viruses or download software to other systems). Diskless workstations use the diskless support facility to obtain an operating system, obtain any configuration information such as a home directory in the CDS to which they are connected, connect to the DFS and perform remote swapping of files. There is no API to the diskless support service and the client application is not aware that it is using a diskless system. All interactions are through the DFS and the diskless protocols (described in the engineering model) which make the remote file access and swapping transparent.

3.4.4 Diskless Support Implementation

The diskless support technology in DCE is based on HP's diskless support technology which uses Internet standards, namely the Internet Boot Protocol (BOOTP) [IAB,94] and the Internet Trivial File Transfer Protocol (TFTP) [IAB,94]. The architecture of the diskless support service is illustrated below:-

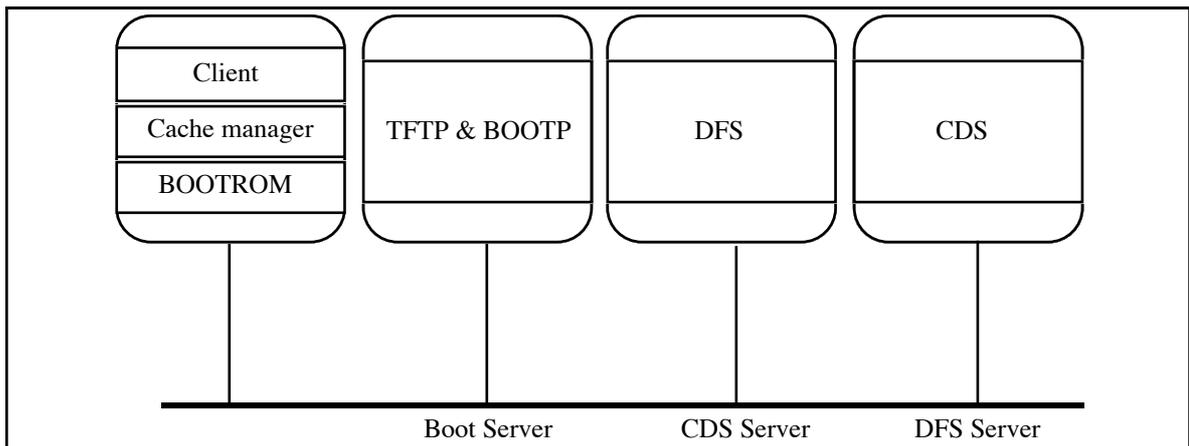


Figure 3.13 - diskless support architecture

The diskless client application sits on top of the DFS cache manager which caches data locally. The BootRom of the diskless client sits in ROM and starts the bootup process for the diskless client when it is switched on, by broadcasting a request to the local cell for a boot server. A suitable boot server returns the necessary information to the diskless client (the boot filename of the client, its network address, the network addresses of the computer the boot and CDS server are located on and the location of the configuration information of the client on the CDS server). The client configuration file is then transferred to the client using TFTP. The diskless workstation may then interact with the DCE system.

3.5 Programming in DCE

3.5.1 Purpose of this Section

The aims of this section are similar to those of the equivalent ANSAware section 2.6. The example illustrated in this section is an implementation in DCE of the same client-server application as in the ANSAware section. It should be noted that for the purposes of brevity and simplicity, the *automatic* binding method has been chosen for this application. The automatic method is also the most natural implementation option in this case. Thus, no explicit reference to the directory service is necessary. The reader is referred to [Shirley,92] for an example of implementations using the implicit and explicit binding methods.

3.5.2 IDL

Interfaces are defined in IDL files. Interface definitions in DCE may be split into an *interface header* and an *interface body*. An interface header is illustrated below:-

```
[
    uuid(A985C864-243G-22C9-D50H-06043C1FCGA3),
    version(1.0)
]
```

In this example, the interface includes an interface UUID and a major version number. The interface body is entered after the interface header:-

```
interface Cabbage
{
    int Add (        [in]    NumberA
                   [in]    NumberB
                   [out]   *Result );
}
```

This defines the *Cabbage* interface with one operation *Add*. Parameters include attributes

indicating the direction that data is passed across the network. Programmers with experience in Ada will notice a similarity between the DCE attributes and similar attributes in Ada. A parameter may be of one of three types:-

- *in* - indicates that a parameter is to be passed to the remote procedure from the client.
- *out* - indicates that a parameter is to be passed from the server to the client. Out parameters must be passed by reference (i.e. must be a pointer or an array) in order to function correctly.
- *in, out* - a combination of both of the above.

Interfaces are compiled by the DCE IDL compiler.

3.5.3 Server Code

Code for server implementation is usually split into two parts in DCE. The first part is the implementation of the operations. The second is the code necessary to initialise the server. These parts are conventionally stored in separate files.

3.5.3.1 Implementing Operations

Code for implementing the server operations is greatly simplified by using the automatic binding method. The file *Cabbage.h* is generated by the IDL compiler and is necessary in order to include the definitions made in the IDL file.

```
#include <stdio.h>
#include "Cabbage.h"
int Add(NumberA, NumberB, Result)
    int    numberA,
          numberB,
          *Result;
{
    *Result = NumberA + NumberB;
}
```

In this case, the code for implementing operations is exactly the same as a normal C procedure⁸.

3.5.3.2 Server Initialisation

The code for initialising a server is illustrated below. It should be noted that this code is fairly generic and is usually the same for most servers. Calls to the initialisation routines usually include a status variable. The return value of this variable may be checked to see whether a call was successful. This code has been excluded in the interests of clarity.

⁸This would *not be the case* for explicit bindings which include the binding handle as an explicit first parameter to remote procedures.

```

#include <stdio.h>
#include "Cabbage.h"
main()
{
    unsigned32      status;
    rpc_binding_vector_t *binding_vector;
    unsigned_char_t *entry_name;
    char            *getenv();

    rpc_server_register_if(
        arithmetic_v0_0_s_ifspec,
        NULL,
        NULL,
        &status );
    rpc_server_use_all_protseqs(
        rpc_c_protseq_max_reqs_default,
        &status );
    rpc_server_inq_bindings(
        &binding_vector,
        &status );
    entry_name = (unsigned_char_t *)
    getenv("ARITHMETIC_SERVER_ENTRY");
    rpc_ns_binding_export(
        rpc_c_ns_syntax_default,
        entry_name,
        arithmetic_v0_0_s_ifspec,
        binding_vector,
        NULL,
        &status );
    rpc_ep_register(
        arithmetic_v0_0_s_ifspec,
        binding_vector,
        NULL,
        NULL,
        &status );
    rpc_binding_vector_free(
        &binding_vector,
        &status );
    rpc_server_listen(
        rpc_c_listen_max_calls_default,
        &status );
}

```

The server firstly registers the *Cabbage* interface with RPC runtime using the *rpc_server_register_if* call. The variable *arithmetic_v0_0_s_ifspec* is an *interface handle* and is predefined by the IDL compiler. The interface handle references vital interface information such as interface UUID's.

The *rpc_server_use_all_protseqs* call informs RPC runtime to obtain information on all available protocol sequences (such as TCP), the host machines and endpoints to create binding information. If several protocol sequences are available, several sets of binding information are stored. It is necessary to obtain a pointer to this information (a *binding vector*⁹) using *rpc_server_inq_bindings*. One function of this routine is to assign endpoints to the server.

The server interface is exported to the directory service using the *rpc_ns_binding_export* call. The call to *getenv* is necessary to obtain the name that the server should export to the directory service. This is defined in a system environment variable. Once a server is exported to the directory service, the *rpc_ep_register* is used to register that the server process running at the endpoints specified in the binding vector is associated with the interface.

Finally, the memory allocated for the binding vector is released when the binding vector is no longer required using *rpc_binding_vector_free*. The server is then told to listen for remote calls using *rpc_server_listen*.

3.5.4 Client Code

In order to implement a DCE client using the automatic binding method, it is necessary to include the header file *Cabbage.h* generated by the IDL compiler.

```
#include <stdio.h>
#include "Cabbage.h"
main()
{
    int Result;
        Add(5,7, Result);
        printf("Result = %d\n");
}
```

The remote operation *Add* is called in the normal C fashion. The correct server is automatically located in the directory service and a binding is established. The *Add* operation is invoked and the result returned.

3.6 Summary

This chapter has not discussed DCE from the ODP perspective as it is essentially an engineering level implementation. Rather, DCE has been discussed from the perspective of services and implementations.

The chapter contains a large amount of information due to the comprehensive set of services provided by DCE. The information detailed here will be used in the following chapter which compares ANSAware and DCE.

⁹This differs from a *binding handle* which is a pointer to one possible binding.

Chapter 4 - Comparing ANSAware with DCE

4.1 Introduction

This chapter provides a comparison of the features and functionality of ANSAware and DCE. The comparison is based around three main sections: architectural issues, properties and services of distributed systems, and programming issues.

4.2 Architectural Issues

This section compares the architectures presented by ANSAware and DCE. Firstly, consideration is given to the motivations behind the respective architectures. This leads to a comparison between the client-server approach adopted by DCE and the object-based approach adopted by ANSAware. Consideration is also given to how ANSAware and DCE fit into the ODP computational and engineering models.

4.2.1 Motivations Behind the Architectures

In order to understand the differences between ANSAware and DCE, it is helpful to consider the motivations behind the two projects.

The ANSA reference model was built as a vehicle for *technology transfer*¹⁰. ANSA is an innovative architecture, designed to provide a long term solution to distributed computing. DCE, however, was designed with commercial interests in mind. The DCE architecture is based on well-understood and tested methods. It is a short-term approach with emphasis on providing a comprehensive set of potentially distributed services rather than innovation.

Although the approach taken by DCE provides service guarantees, the architecture must evolve to remain competitive. A fundamental area in which DCE lacks innovation is in its approach to *how* it provides services. This is discussed in the following section.

4.2.2 From Client-Server to Object-Based

The short-term approach taken by DCE has led to an architecture based on the client-server paradigm. Thus, DCE users and applications see a system organised as a number of potentially distributed server processes offering particular services. DCE clients may make requests to servers via remote procedure calls.

This paradigm is suitable for providing a coarse level of distribution but provides limited functionality and flexibility. For example, the client-server paradigm offers little abstraction over distribution. There is no concept of a centralised repository of available services such as a trader. Furthermore, there is no method of classifying services into hierarchies (i.e. type and context hierarchies) and it is difficult to visualise migrating services from one address space to another.

The innovative approach taken by an object model provides the abstraction, functionality and flexibility lacking in the client-server approach. Moreover, objects are a natural way of modelling distributed systems (i.e. distributed components communicating via messages to well-defined interfaces with locally implemented procedures). Objects support heterogeneity as messages between objects depend on the object interface only. Objects also support autonomy as the underlying implementation of a service may be altered transparently providing that the corresponding interface is maintained.

Moving to an object-based paradigm from a client-server paradigm is not trivial but

¹⁰Technology transfer refers to the process of an enterprise moving from one combination of hardware and software components to another.

provides additional benefits. To illustrate this further, we will revisit the well-defined object model of ANSAware. The computational objects in ANSAware allow us to support abstraction and have the benefit of integrating into the object-oriented design methodology. For example ANSAware objects may be defined by templates and instantiated. Furthermore each object is typed and may be grouped into a class with other objects of the same type. Objects may be exported to and imported from the trader according to their type and attributes providing a central repository of services. The trader also facilitates code reuse (through multiple clients importing the same service).

An object model uses invocation as opposed to the remote procedure call for making requests. Invocation provides clear advantages over the remote procedure call. It does not imply a particular method of interprocess communication. Invocations can be optimised for local communications (for example ANSAware uses named UNIX pipes for local communications). Alternatively, invocations can be implemented using distributed virtual memory techniques where functionality is copied locally and executed as a local procedure. Invocations may also transparently support object persistence and migration. These abstractions are not currently supported by DCE.

In addition, ANSAware objects are of finer granularity than DCE. ANSAware capsules support many objects within an address space while DCE clients and servers are typically processes. Thus ANSAware objects optimise the use of system resources and minimise the overhead of object passivation and migration.

4.2.3 More on Integrative Standards

The positioning of DCE, ANSAware and the various integrative standards is illustrated in figure 4.1.

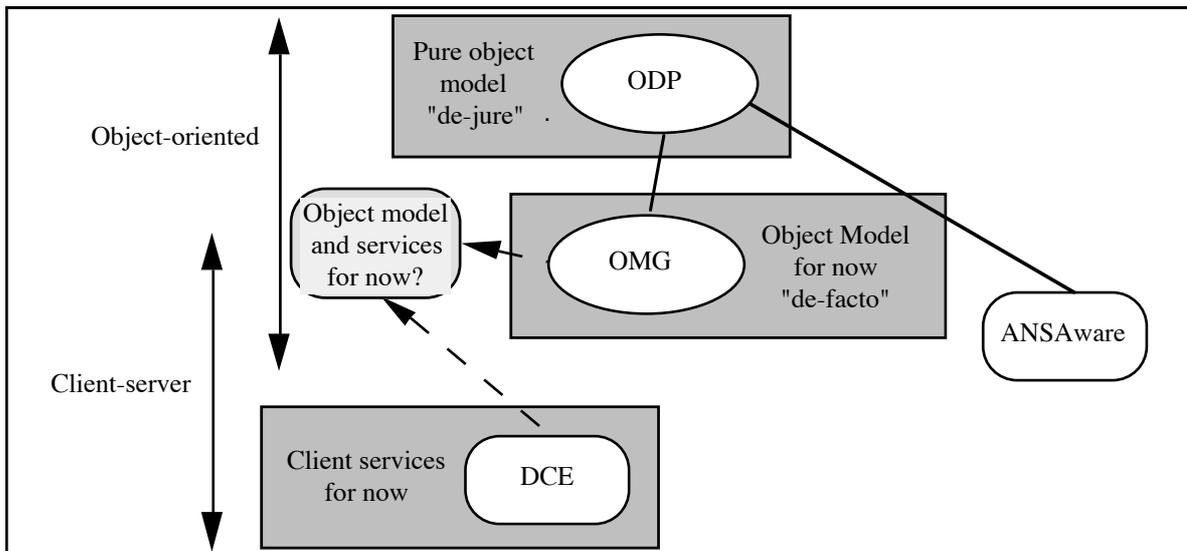


Figure 4.1 - integrative standards

This figure illustrates the relationship between the efforts of the Object Management Group and Open Distributed Processing. The OMG integrative standard is a present day de-facto standard. This can be contrasted with the pure approach taken by ODP¹¹ which cannot be currently implemented for reasons of technology limitations and due to its draft status.

In order to fit DCE into the ODP standardisation effort, it is necessary to provide it with

¹¹The reader should note that there are currently conflicts between the ODP object model as defined by the computational viewpoint and the OMG object model e.g. in the areas of terminology such as the definition of *subtypes*.

an object model. The OSF has stated that it intends to move towards ODP compatibility in the future. This implies that DCE will have an object model in the long-term. A more immediate solution to this problem is to integrate the CORBA standard from OMG into the DCE architecture (see section 5.4). This will evolve DCE into a comprehensive distributed architecture providing services through a well-defined object model.

4.2.3 ODP Viewpoints

In terms of the ODP viewpoints, DCE is an implementation providing engineering-level functionality. In contrast, ANSAware is concerned with the computational, engineering and technology viewpoints. The amount of mature work on the enterprise and information viewpoints in either architecture is insignificant¹².

4.3 Properties and Services

The remainder of this chapter will compare ANSAware and DCE from the perspective of their *properties* and *services*. Firstly we will define the meaning of these terms and then use them as a basis for comparison.

4.3.1 What do you need from a distributed system?

According to Schroeder, distributed systems can be described in terms of the properties and the services that they offer [Schroeder,93]. He classifies the properties of distributed systems under the heading as global names, global access, global security, global management and global availability. Furthermore, he classifies the services offered by distributed systems into names, remote procedure calls, user registrations, time, file and management. Additional services are defined which are specific to appropriate applications. These are records, printers, execution, mailboxes, terminals and accounting. We will adopt these headings for the purposes of this section.

4.3.1.1 Some Definitions - Properties

A *global naming* scheme allows the same textual naming conventions to be used anywhere in a distributed system. Thus, in a UNIX based system, the same pathname may be used to reference specific files from different machines.

Global access provides a user with the similar functionality anywhere in a distributed system. The only tradeoff which may be perceivable to such a user would be a drop in performance when accessing data from a remote site. Thus a user may access a particular file or software package on a particular system from any location. Global access implicitly raises the problem of providing users with a coherent view of a system. Two or more users collaborating over a particular file, for example a textual document or a data file of statistical calculations, should have the same view of a document at any particular time. Thus, when *user 'A'* saves changes to a document in the United States, the updated state of the document should be immediately visible to *user 'B'* in the United Kingdom.

Global security provides authentication and authorisation consistently across a distributed system, allowing a user to log on to a particular machine with the correct privileges and access rights. It allows users to confer access rights such as read or write access to particular users or groups of users. It also provides users with guarantees that a communication is secure (i.e. free from eavesdropping and encrypted).

Large distributed systems require a *global management* policy which allows management tasks to be carried out by several individuals (with the correct access privileges). All management tools should have a common interface. Thus, a manager can perform an action on a system (such as a workstation), or a group of workstations remotely.

Global availability provides a user with some service guarantees, usually by replicating

¹²Note that the ANSA architecture has considered these issues in greater detail.

services. This provides a level of redundancy and therefore fault tolerance. As long as the amount of failures does not exceed the redundancy then services are guaranteed.

4.3.1.2 Some Definitions - Services

Names provide access to a replicated distributed database of global names (i.e. a name service). This must be integrated with existing components of a distributed system. Examples of global names include access control lists, services, machines and files.

Remote procedure calls allow transparent invocation of local and remote procedures. This service supports asynchrony i.e. does not force a calling client to block while an RPC is sent.

User registrations relate to *security issues*. Users should be registered in a user database to allow authentication to take place. Authenticated users are issued with a privilege certificate which contains their credentials. These specify which services and actions a user is permitted to access.

A *time service* provides a global notion of time within a distributed system.

A *distributed file service* provides a global distributed file system. This should be replicated for availability. The file system should use the name service to provide a global naming scheme allowing transparent access of local and remote files in a distributed system.

Management of distributed systems is an increasing problem as systems grow and associated management costs soar. Each service should have associated management operations.

Records provide access to indexed database records with failure support (transactions) and record locking. Printer job control and scheduling should be provided with support for relevant formats such as postscript.

Execution allows a program to run on any machine (or group of machines) from a collection of heterogeneous machines on a network¹³. The concept of execution includes the efficient scheduling of jobs on machines, support for failures, priorities and deadlines.

4.3.2 Properties

4.3.2.1 Global Names

This section considers the key differences in the approaches to global naming taken by ANSAware and DCE. ANSAware provides a global naming scheme via a namespace of federated traders. Service queries are generally via the trader, using the import and export facilities. This approach is in contrast to DCE which provides an X.500 based global naming scheme linked to a local cell directory service. Services are referred to by explicit global pathnames which are passed as parameters to other DCE services.

The ANSAware trader service may be used to perform attribute based lookups (a *yellow pages* service) while DCE provides a name lookup (a *white pages* service), and a yellow pages style of lookup via group and profile facilities. The yellow pages facility provided by DCE is not as comprehensive as that defined by ANSAware. The latter service includes the concepts of namespace contexts and type spaces.

While the naming schemes provided by DCE and ANSAware address the issue of extensibility¹⁴, their approach to scalability¹⁵ is different. DCE supports scalability by providing cells to partition the distributed system into smaller domains which can be

¹³Subject to access restrictions.

¹⁴Extensibility refers to whether it is possible to add more nodes to a system.

¹⁵Scalability refers to the amount of nodes that it is possible to add to a system.

managed by a local cell directory service. If a cell becomes overcrowded, it can be partitioned into further cells¹⁶. ANSAware supports scalability via the process of federated trading.

4.3.2.2 Global Access

ANSAware provides access to services via the object model defined in the computational viewpoint. This well-defined object model allows a user to import an interface from the trader and invoke operations provided by the underlying object regardless of its location. This contrasts with the paradigm presented by DCE which provides engineering-level procedure calls to the RPC runtime library. These calls require explicit registration of low-level information such as endpoints.

ANSAware provides the concept of an interface reference for referring to a specific interface which is obtained dynamically via the import statement. DCE uses an interface UUID and optionally object UUID's to identify a particular interface. The interface UUID is defined statically at compile-time in the IDL file.

Bindings in DCE may be automatic, implicit or explicit. DCE allows various details of implementation to be hidden using the automatic and implicit methods while binding handles can be explicitly managed using the explicit method. ANSAware provides one type of binding which is roughly equivalent to the explicit binding facility of DCE. Thus the user of ANSAware is always aware that they are accessing potentially distributed services.

4.3.2.3 Global Security

While the ANSA architecture has given some consideration to security, this work has not currently fed into ANSAware. DCE incorporates a security service which is integrated with the other services. Security in DCE is comprehensive providing encryption, verification, authentication and authorisation mechanisms.

The lack of a security service in ANSAware is related to the differing philosophies behind the design of ANSA and DCE. In order for a system to be commercially viable, it is essential that the enterprise it serves can be confident that the system is secure. Since DCE is designed as a commercial strength product, security is a high priority. In terms of innovation and technology transfer (i.e. ANSAware), security, although important is not as essential as a comprehensive object model.

4.3.2.4 Global Management

Both ANSAware and DCE provide graphical tools to ease the burden of management in a distributed system. ANSAware provides graphical interfaces (usually X-Windows based) to key services such as the trader. ANSAware also provides node manager facilities for managing the resource of a node. DCE provides a range of tools for manipulating services such as the directory service, the security service and remote procedure call service.

Further efforts are being made in the area of global management by the Open Software Foundation. These efforts have recently resulted in the *Distributed Management Environment* (DME). The architecture of DME is illustrated in figure 4.2.

¹⁶The next release of DCE (version 1.1) will also support cell hierarchies.

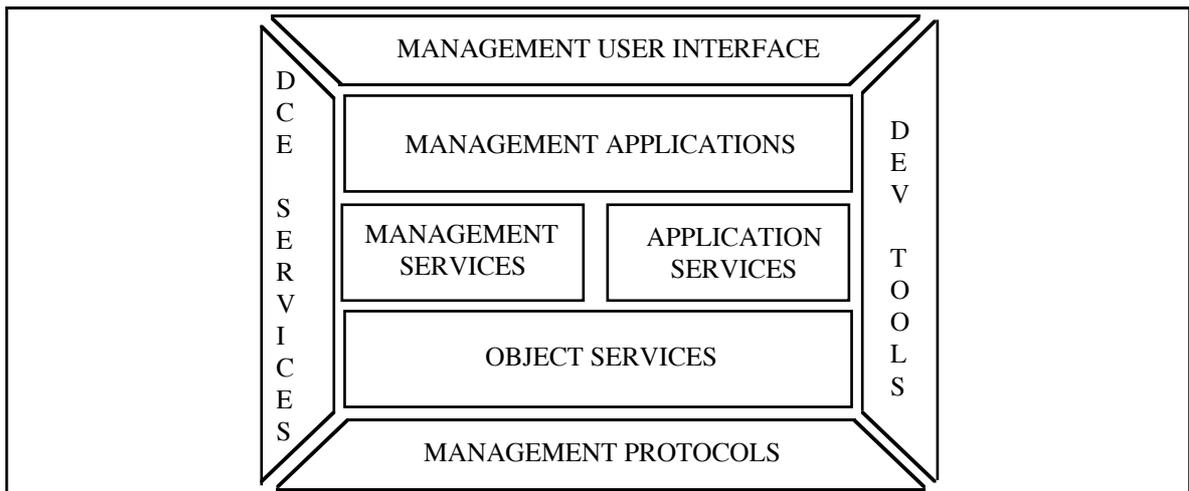


Figure 4.2 - DME architecture

DME recognises the problems of administration created by environments such as DCE which may run over many heterogeneous systems each with a different management scheme and possibly varying management tools. DME aims to solve the problem of soaring management costs by providing a consistent set of management tools. This framework allows the management of systems and networks via a graphical interface.

Typical functions provided by DME include the management of software installation, peripherals (such as printers), application services and software licensing.

DME is integrated with the DCE services. An important point to note is that DME includes *object services*, such as those provided by CORBA. DME is also integrated with many of the existing management standards including ISO standards and TCP/IP. Further information on DME may be found in [DME,91].

4.3.2.5 Global Availability

ANSAware provides increased availability of services using the group execution protocol GEX for the replication of services. DCE tackles the problem of availability by replicating its services within cells. Typical replicated services in DCE include the security service, RPC and the cell and file servers.

The trading service in ANSAware is not replicated for availability. This type of replication must be explicitly implemented (if required). Replication in ANSAware is only for federation (i.e. the dividing of the trading namespace into smaller components). The DCE name service is a highly replicated distributed database. This is a considerable strength when compared to ANSAware. Replicating the name service across several nodes in a cell provides increased performance and improved fault tolerance.

4.3.3 Services

The following table lists the services of distributed systems and indicates the extent to which ANSAware and DCE fulfil them:-

	ANSAware	DCE
Names	Federated trading, yellow pages service.	Replicated distributed database, mainly white pages service, some yellow pages service.
RPC	REX RPC.	DCE RPC.
User Registrations	None.	DCE security service providing authentication and authorisation. Integrated with other services.
Time	None.	Distributed Time Service providing a global notion of time.
Files	None. ANSAware users see a persistent object store as opposed to a file system. The underlying file system provided by the host operating system (such as NFS) is accessed transparently when required. Note that it is still possible to use the underlying file system if necessary.	Distributed file service integrated with the directory service. DCE provides a replicated file system to improve availability.
Management	X-Windows based tools for interacting with the trader (e.g. viewing files). Node manager for managing the resources on a node.	Management functions integrated with each DCE service. Management tools for interacting with the directory, security and RPC services. Recent release of DME as a global system management tool.
Records	Arjuna transaction service.	Encina database service.
Printers	No direct support. Possible implementation as a service.	No direct support. Possible implementation through object UUID's. Recent release of DME providing printer functionality.
Execution	Support through trading. No support for priorities or deadlines.	Support through the directory, file and security services. No support for priorities or deadlines.
Terminals	Operating system support.	Operating system support.
Accounting	None.	DME.

4.4 Programming Issues

DCE uses a client-server paradigm with the remote procedure call as the method of communication. ANSAware uses an object-based paradigm and uses invocations as opposed to remote procedure calls.

4.4.1 Comparing IDL's

The key differences in IDL's between ANSAware and DCE are listed below:-

Abstraction

The IDL provided by ANSAware provides a high level of abstraction. This includes support for a range of IDL-specific data types such as sequences [APM,93] and interface references to uniquely identify interfaces. ANSAware IDL also supports several "common" data types¹⁷. DCE IDL provides less abstraction than ANSAware and resembles C in syntax. It provides support for the common data types but does not support the range of IDL-specific data types provided by ANSAware.

Support for Pointers

DCE IDL provides explicit support for pointers and strings. ANSAware IDL does not.

Unique Identifiers

DCE supports UUID's to uniquely identify interfaces. Similar functionality is provided by ANSAware through interface references but not in the IDL.

Version Control

DCE IDL includes version numbers for version control. This feature is not supported by ANSAware.

Support for Operations

Support for operations in ANSAware is provided by named operations with input parameters and return types. DCE provides an Ada-like syntax supporting operations with in, out and in-out parameters. Operations in ANSAware may be invoked using the announcement and interrogation syntax. DCE provides default and maybe semantics.

Inheritance

ANSAware supports the inheritance of data types and operations from other IDL's. DCE only supports the inheritance of data types.

4.4.2 Comparing Application Code

The following list details the key differences between application programming in ANSAware and DCE:-

Distributed Processing Language

ANSAware embeds prepc statements into C source code¹⁸. This has the advantage of making embedded statements easily identifiable and provides abstraction. DCE uses libraries of C functions providing services. This makes DCE statements difficult to distinguish from other source code and does not provide any abstraction over the basic engineering concepts of distribution.

Concurrency

Concurrency is available in ANSAware through threads, announcements and voucher operations [APM,93]. DCE also provides a thread package, broadcast semantics and maybe semantics (maybe semantics are equivalent to announcements in ANSAware).

¹⁷For example integers, characters and booleans.

¹⁸Other source languages may be used as long as they can mimic C calling conventions.

Registering and Locating Services

ANSAware services export their interfaces to the trader while DCE requires a much lower level of interaction. It is necessary to register with the endpoint mapper, RPC runtime and the name service.

Bindings

ANSAware offers a single binding method while DCE offers a selection, namely automatic, implicit and explicit. The explicit binding in DCE is roughly equivalent to the binding provided by ANSAware. The automatic binding method in DCE provides a high level of transparency over distribution, as demonstrated by the client program in the cabbage example (section 3.5.4). It is not possible to achieve this level of transparency in ANSAware.

Miscellaneous

In addition to the above, DCE provides pipes for the rapid transfer of bulk data and context handles which allow servers to maintain client state information. This limits data transfer overheads. DCE also provides security, time, and file operations.

4.5 Summary

This chapter has compared ANSAware and DCE from the aspect of properties and services. The comparison has indicated a number of differences between the two architectures.

The key differences between ANSAware and DCE are that ANSAware provides a well-defined object model and is based on an innovative architecture. However, the architecture provides a limited set of services. In contrast, DCE provides a comprehensive set of services for distributed systems, has a broad industrial backing and is based on well understood methods. Key services provided by DCE that do not appear in ANSAware include the file, time and security services.

The DCE architecture is based on the client-server paradigm and this makes it inflexible. It is important that this issue is addressed in the near future. It is generally accepted that DCE must evolve an object model in order to remain competitive. Integrating CORBA with DCE is a possible solution (this would potentially involve mapping the CORBA interfaces on to the underlying DCE services). CORBA is an attractive solution as it has a well-defined object model and implementations of the architecture are currently available (unlike ODP). Integrating CORBA with DCE is discussed further in section 5.4.

Chapter 5 - Concluding Remarks

5.1 Introduction

This report has looked in detail at the architectures of ANSAware and DCE.

Chapter 1 provided an overview of the three major integrative standards; the Open Distributed Processing Draft Standard from the International Standards Organisation (ISO/ODP), the Distributed Computing Environment from the Open Software Foundation (OSF DCE) and de-facto standards endorsed by the Object Management Group (OMG). Chapter 2 then described a framework for creating distributed computer systems called the ANSA reference model. From this chapter, it emerged that the model has similarities to concepts defined in ISO/ODP such as a well defined model of objects and their interactions, services, and trading. ANSAware, a partial implementation of the concepts defined in the ANSA reference model was also described.

Chapter 3 provided an overview of the major goals of OSF DCE, introducing the concept of interoperability. This overview was followed by a study of the programmer-oriented fundamental DCE services, the user-oriented data-sharing services and their respective implementations. Finally, Chapter 4 compared ANSAware and DCE. In the present chapter, we summarise the major results of this comparison and present some conclusions about how convergence between integrative standards may be realistically achieved. We also consider some of the emerging areas of research in distributed systems (i.e. Mobility, Multimedia and CSCW) and the challenges that they present.

5.2 Major Results

The major results of the comparison are as follows:-

- *Differing Goals* - ANSAware has been designed for technology transfer (i.e. to enable an enterprise to move to a distributed environment), while DCE has been designed for commercial acceptance.
- *Objects vs Services* - ANSAware provides an object-based approach while DCE is service-oriented. DCE would benefit from an object model whose functionality maps to the underlying DCE services. DCE cannot be compared using ODP as a basis as it is an engineering-level implementation (furthermore, DCE is not defined by a comprehensive engineering language).

5.3 Other Results

Other results include:-

Trading vs Name Service

The ANSAware object model facilitates global naming by providing the means for a trading service with a yellow pages style of interaction. This attribute-based style of approach allows a comprehensive method of object interaction. This is complementary to the X.500 based naming service of DCE which is a predominantly white-pages service.

Services

DCE provides security, file, time and diskless support services over and above those available in ANSAware. ANSAware provides the node manager and the factory and notification services.

Management

Name spaces in ANSAware are managed by the federation of traders. DCE manages a collection of autonomous cells through a global namespace. ANSAware provides little in the way of management tools, apart from a small number of basic facilities for management of objects such as the trader. DCE provides tools for the management of the directory service, RPC's, and security. Furthermore, tools built specifically for management of large distributed systems such as DME are available which use DCE.

Performance

DCE is optimised for performance at the intra-cell level, through replicated services such as the directory service. This provides availability and robustness. ANSAware federation does not optimise performance.

Programming Issues

ANSAware provides a high level of abstraction in terms of source code, both with IDL and DPL files. DCE provides a set of procedure calls in the host language (currently C) to the services that it supports. DCE calls provide much more functionality than ANSAware due to the increased number of services available.

5.4 Looking Ahead

This section takes a brief look at why convergence between the relevant integrative standards is necessary, whether convergence is possible and how it can be achieved. Finally, consideration is given to the new challenges that lie ahead in distributed systems.

5.4.1 Why is convergence necessary?

The aim of integrative standards is to address the key issue of *interoperability*. Heterogeneous hardware and software platforms populating the distributed world should be able to interact through uniform interfaces using uniform concepts. Currently, distributed systems conforming to a particular integrative standard achieve these aims. However, interoperability across distributed systems conforming to *different* distributed systems is not guaranteed. Only a limited level of "openness" has been achieved.

Similarities exist between integrative standards. At the conceptual level, both OMG and ODP have a well-defined object model and method of service location (i.e. the trader in ODP and the interface repository in OMG). However, conflicts in terminology exist (for example an "interface" in ODP is known as a "type" in OMG and the term "object" is meaningless in DCE). DCE provides a powerful set of services and yet lacks the sophistication to migrate it's services to ODP or OMG platforms.

Increased convergence between integrative standards will address these issues. A clear requirement exists for a more uniform conceptual model (i.e. how users of distributed systems and distributed applications perceive their environment). Convergence should lead to a way of modelling distributed systems that exploits the best benefits of each standard.

5.4.2 Achieving Convergence

This report has concluded that an object based approach is preferable for providing distributed services. Hence, there is a requirement for DCE to support an object model. This may be achieved by integrating OMG CORBA with the DCE architecture.

Figure 5.1 illustrates a gateway mapping OMG to two disparate worlds, namely DCE and SUN. Mapping OMG to DCE potentially involves mapping OMG interfaces to the underlying DCE function calls allowing communication with DCE. Mapping OMG to the SUN Open Network Computing (ONC) remote procedure call interface would involve a similar process.

The potential exists to integrate other transport and remote procedure call protocols with OMG standards. Appropriate gateways would be necessary to allow communication between different platforms. This flexibility is important as OMG are committed to an open approach. DCE is a strong contender for providing distributed services to support OMG standards.

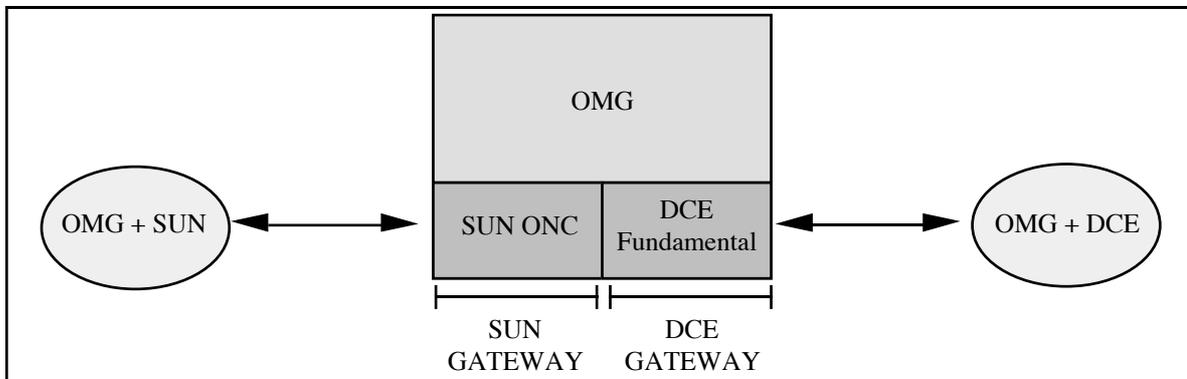


Figure 5.1 - Integrating OMG standards with DCE and SUN

The OMG integrative standard offers a realistic object model that can be supported by the majority of today's object-oriented systems. Only the fundamental DCE services are currently illustrated in figure 5.1 as OMG standards do not currently address the issues surrounding data sharing services such as diskless support and distributed file systems. Potentially, OMG standards would co-exist with a file system by mapping persistent objects to the underlying file system. In terms of diskless support, persistent objects would still be mapped on to the underlying file system but the file would reside on the disk of a remote machine.

The OMG approach can be contrasted with the pure approach taken by the more long term ODP integrative standard which can only be partly realised by a small group of specialised platforms such as ANSAware. In this way, the OMG integrative standard can be considered as a 'fast track' to ODP. Cordial relations exist between ISO and OMG and discussions are currently taking place regarding the convergence of ODP and OMG.

5.4.3 New challenges

As users become increasingly familiar with a particular system, they begin to accept the facilities provided as normal and look for more sophisticated functionality. This is true of distributed systems. Distributed computing continually faces new challenges as increasing numbers of enterprises adopt a distributed approach.

The dynamic nature of distributed systems may be illustrated by considering the impact of *multimedia*, *mobility* and *computer supported cooperative work (CSCW)* on distributed systems. These research topics extend the distributed paradigm and provide new research challenges.

5.4.3.1 Multimedia

One class of applications that has sparked research and commercial interest is that of distributed multimedia applications. Such applications embody the concept of *continuous media*. In contrast to static media (e.g. still text and graphics), continuous media has a temporal dimension (viz. video and audio). Examples of multimedia applications include desktop conferencing, video on demand and interactive teaching applications.

Distributed systems must provide support for continuous media if they are to be used as an underlying basis for distributed multimedia applications. We believe that distributed systems must address the following requirements:-

Explicit Support for Continuous Media

Explicit support is required to provide a continuing commitment for the real-time characteristics of continuous media. This includes a method of resource reservation to reserve necessary resources such as networks and memory in advance. New programming abstraction are necessary in order to model and capture these requirements.

Quality of Service (QoS) Support

Quality of service support is necessary in order to provide end-to-end guarantees of service and to support tailorability. Distributed systems need to be tailorable in order to provide support for different styles of continuous media. Voice quality audio, for example, requires significantly less network bandwidth than full colour full frame video. Applications may make a quality of service request in order to provide the guarantees that they need. If it is not possible to meet that request, a process of negotiation must take place between the application and the underlying distributed system. This may result in a degradation of quality (e.g. full colour video may be reduced to black and white if the bandwidth required is not available).

Synchronisation Support

Multimedia requires a range of synchronisation mechanisms, such as support for real time events (e.g. displaying a dialogue box when a certain video frame number is reached) and for continuous synchronisation (e.g. lip-sync).

Group Communication Support

Multimedia applications tend to support groups of collaborating users. A typical example of this class of multimedia application is a video conferencing application. Support for groups includes facilities to message groups (e.g. control messages) and to transmit continuous media to groups.

Further information on the requirements of multimedia may be found in [Davies,93].

The suitability of ANSAware as a support platform for multimedia has been demonstrated by the development of the *base services* at Lancaster University [Coulson,93]. DCE has not yet been used as a support platform for multimedia although research has been carried out into the implications of continuous media support for multimedia [Adcock,93]. This research indicated that DCE could be a support platform for multimedia with the introduction of a set of multimedia services into the architecture.

5.4.3.2 Computer Supported Cooperative Work

Computer Supported Cooperative Work (CSCW) has emerged as an identifiable research area which focuses on the role of the computer in group work. The exact meaning of the term CSCW has undergone considerable debate [Bannon,91][Robinson,91]. Most authors however agree on the following principles:-

- i) *cooperation* - work is a *cooperative* activity, generally involving groups of people interacting to achieve common goals, and
- ii) *support* - the designers of supporting computer systems¹⁹ must address this cooperative nature of work.

Perhaps the most striking feature of CSCW is the inter-disciplinary nature of the work. Inter-disciplinary research in computing is not new. For example, human computer interaction has developed as an inter-disciplinary subject involving psychologists and

¹⁹The term *groupware* has emerged to signify software systems which address the cooperative nature of work.

computer scientists. In CSCW, contributions to the field have been made by such diverse disciplines as computer science, psychology, sociology, economics and organisational studies. Perhaps the most notable contribution has come from sociological traditions. In particular, the field of ethnographic analysis has emerged as a particularly prominent technique to aid the development of groupware systems [Hughes,92].

The majority of CSCW applications are distributed and thus depend on the facilities provided by existing distributed systems platforms. They also typically require *open* solutions to distributed processing in order to support co-operative work between different departments, sections or even organisations with each likely to have different computer equipment, operating systems, work practices and management policies. Research has shown CSCW will have an impact on distributed systems. This is typically because many CSCW applications have stringent requirements for the underlying infrastructure, e.g. in terms of multimedia communication or real-time interaction. We will discuss briefly a selection of the prominent areas of activity in CSCW:-

Workflow systems

Workflow systems are based on the notion of workflow or activity. These systems have developed from previous considerations of office automation and adopt a process oriented perspective on group work. Cooperative work is viewed as items of work flowing between a number of activities. Message exchange is the predominant means of representing this flow of work.

Multimedia and desktop conferencing systems

Desktop conferencing systems stem from the merging of workstation technology and real-time computer conferencing. Such systems enable groups of users to simultaneously interact with one or more applications; voice and video facilities are also often provided. Two main approaches have been identified in the CSCW community, i.e. collaboration-transparent and collaboration-aware conferencing. Collaboration-transparent systems enable existing applications to be viewed in a group setting (examples include Rapport [Ahuja,88], SharedX [Garfinkel,89] and MMConf [Crowley,90]). In contrast, collaboration aware solutions provide facilities to explicitly manage the sharing of information, allowing sharing to be presented in a variety of different ways to different users.

Multi-user hypertext

Multi-user hypertext systems are a significant focus of research in CSCW because of their ability to support flexible structuring mechanisms. Within such systems, the hypertext document (or network) is constructed by a number of users adding nodes to the network in an independent manner. Facilities must then be provided to deal explicitly with the conflicts inherent in this process of interaction. Systems following this line of development include Intermedia [Garret,86], Notecards [Trigg,86] and Sepia [Haake,92]. *Co-authoring systems* are one class of cooperative systems which apply the principles of hypertext technology in a cooperative setting. The Quilt system [Fish,88] developed at Bell Communications is representative of the general principles used by most co-authoring systems.

The cooperative nature of group working means that there is a requirement to be aware of concurrent activity. Such a requirement appears to conflict with the ODP concept of transparency. Thus, ODP must evolve in order to effectively support group working. These issues are discussed further in [Blair,93] which considers the potential impact that CSCW will have on ODP. This study has highlighted the need for more cooperation aware policies for concurrency control, security and management.

5.4.3.3 Mobility

Developments in portable computer technology and mobile communications have led to increased interest in mobile systems. Mobility involves connecting portable nodes (e.g. workstations or PC's) by wireless means to a fixed network. Typical wireless methods of connection include infra-red and radio links.

Considering the advances in the supporting technology for mobility, extending mobile systems to support CSCW is now becoming feasible [Adcock,94]. In modern organisations, significant numbers of workers are mobile, either travelling between sites or carrying out work at remote locations. These technological advances have the potential to revolutionise work practices for such employees. For example, the utilities industries in the UK are looking very closely at the potential impact of mobile support for field engineers [Davies,93a].

At present, there is very little work in this area. Most research has a technological focus. For example, there is considerable research into communications technologies to support mixed media (voice and data) into the field. Research is also being carried out into high bandwidth links such as microwave connections [O'Reilly,93]. Researchers are also looking at issues such as disconnected file systems and addressing mechanisms for mobile computers [Satyanarayanan,93]. There has been much less work on the social impact of mobility. It is likely that mobile computing will influence both CSCW and integrative standards.

With the relative immaturity of mobile computing, it is difficult to assess its likely impact on integrative standards. Several potential problem areas can however be highlighted:-

i) Techniques for transparency

Integrative standards must address the issue of providing transparent access to services from mobile hosts. With the limited bandwidth of radio communications, this means that new techniques will be required, for example, to cache significant portions of the data on the mobile computer. Care must also be taken to maintain consistency if data is shared across several mobiles, e.g. in a conference situation.

ii) Impact of disconnection

Mobile communications are characterised by their peculiar error characteristics, for example users are likely to be disconnected for significant periods of time. It is important that quality of service requests can specify accepted levels of disconnection and that quality of service management can monitor and react to such circumstances. It is also important to trace the impact of disconnection on other dependent services.

iii) Levels of disconnection

Over a period of time, connection may vary from being disconnected to being partially connected (through a radio network) to being fully connected (through a high speed network). This is likely to have a significant impact on techniques for configuration management and binding support. It is also likely that services will take advantage of higher levels of connection to perform bulk updates, e.g. of cached data.

iv) Communications architectures

It is clear that new protocols are required to cope with the characteristics of mobile communications. Particular attention is required by the need to support multicast communications across radio systems. New techniques are also required for mixed media traffic. Existing mobile communications architectures are tailored towards the support of voice traffic. They are not well suited to the bursty style of data traffic generated by many distributed applications.

The impact of mobility for ODP is currently being investigated in the MOST Project (Mobile Open Systems Technologies for the Utilities Industries) involving Lancaster University, EA Technology and APM [Davies,93b]. Research is also being carried out into the impact of mobility on DCE. It is clear that the existing DCE file system does not have the required functionality to support mobility. Furthermore, extensions to existing file systems for disconnected operations such as CODA [Satyanarayanan,93] consider the aspect of full disconnection and do not address the concept of partial connections.

5.4.3.4 Future Work

Ongoing research by the authors is addressing the issue of exploiting partial connections and developing a generalisation of a mobile file system. This will allow the seamless integration of fully/weakly connected and disconnected environments. Such a file system will give the impression to the programmer/user at the ODP computational level [ISO,92] of a normal file system such as NFS or AFS augmented with additional facilities. One such facility is a method of specifying the Quality of Service (QoS) constraints that an application may require (for example to allow support for continuous traffic such as audio and video over high speed networks and for measuring the strength of connections over a mobile link).

The provision of these additional services has an impact at the ODP engineering level as the system will operate using multiple servers over different networks and levels of connectivity, including high speed networks such as ATM and B-ISDN. To summarise, issues that must be considered in order to provide such a file system within DCE include:-

- *Cost* - The cost of providing QoS.
- *Visibility* - The level of QoS visibility at the computational level.
- *Strategy* - Developing strategies to deal with QoS considerations in the face of variable levels of connectivity over a connection.
- *Implementation* - Implementation considerations such as the changes necessary to existing file systems.
- *Integration* - Integration issues with DCE (e.g. do we modify the existing file system or replace it?).

5.5 Summary

This section has summarised the major conclusions reached by the ANSAware/DCE comparison and the other conclusions reached separately. The section then moved on to future issues such as the possibility of a convergence of integrative standards. Three innovative classes of distributed applications that are currently under research, namely CSCW, multimedia and mobility have been briefly mentioned. The implications of such work on the architectures of ANSAware and DCE have been considered.

Glossary

ACSE	Association Control Service Element. A protocol element (OSI application layer) concerned with establishing and releasing a logical connection between two session layer components.
ISO/IEC	International Standards Organisation/International Electrotechnical Commission. The IEC cooperate with the ISO for for Information Technology standards).
OSI	Open Systems Interconnect. A seven-layer communication reference model by the ISO.
ROSE	Remote Operation Service Elements. A protocol element (OSI application layer) concerned with initiating operations and receiving results from a remote session element.
SNA	Systems Network Architecture. A proprietary protocol architecture for communication between IBM machines.
TCP/IP	Transmission Control Protocol/Internet Protocol. A connection-oriented protocol for data transport (OSI transport and network layers).
X/Open	

References

- [Adcock,93] P. Adcock, N. Davies and G.S. Blair: Supporting Continuous Media in Open Distributed Systems Architectures, *Proc. International DCE Workshop, Lecture Notes in Computer Science*, Karlsruhe, Germany, Vol 731, 7-8 October 1993, Pages 179-191.
- [Adcock,94] P. Adcock and A. Friday and K. Cheverst: Literature Survey of the State of the Art in Mobile Systems, *Document Still Under Development*..
- [Ahuja,88] Ahuja, S.R., Ensor, J.R. and Horn, D.N.: "The Rapport Multimedia Conferencing System", *Published in Allen R.B. (ed), "Proceedings of the Conference on Office Information Systems (COIS'88)"*, March 1988, Palo Alto, California.
- [APM,93] Architecture Projects Management Ltd.: The ANSA Application Programmers Guide Release 4.1, Architecture Projects Management Ltd., Cambridge, U.K., February 1993.
- [APM,93a] Architecture Projects Management Ltd.: The ANSA System Programmers Guide Release 4.1, Architecture Projects Management Ltd., Cambridge, U.K., February 1993.
- [APM,93b] Architecture Projects Management Ltd.: The ANSA Reference Manual Release 01.00, Architecture Projects Management Ltd., Cambridge, U.K., February 1993.
- [Bannon,91] Bannon, L. and Schmidt, K.: "CSCW: Four Characters in Search of Context", *Published in J.M. Bowers and S.D. Benford (eds): "Studies in Computer Supported Cooperative Work. Theory, Practice and Design"*, North-Holland, Amsterdam, 1991.
- [Bentley,92] Bentley, R., Hughes, J., Randall, D., Rodden T., Sawyer, P., Sommerville, I. and Shapiro, D.: "Ethnographically Informed Systems Design for Air Traffic Control", *Proceedings of CSCW'92*, Toronto, November 1992.
- [Blair,93] Blair, G.S. and Rodden, T.: "The Challenges of CSCW for Open Distributed Processing", *Presented at the International Conference on Open Distributed Processing*, Berlin, Germany, September 1993.
- [Coulson,93] G. Coulson, G.S. Blair, N. Davies and N. Williams: "Extensions to ANSA for Multimedia Computing", *Computer Networks and ISDN Systems*, Vol. 25, 1992, Pages 305-323.
- [Crowley,90] Crowley, T., Baker, E., Forsdick, H., Milazzo, P. and Tomlinson, R.: "MMConf: An Infrastructure for Building Shared Applications", *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW '90)*, Los Angeles, CA, October 7-10 1990, ACM press, ISBN 0-89791-402-3.
- [Davies,93] N.A. Davies and J.R. Nicol: A Technological Perspective on Multimedia Computing. *Computer Communications* Vol. 14 No. 5, June 1991, Pages 260-272.
- [Davies,93a] Davies N., Blair G.S., Friday A., Cross A.D. and Raven P.F. (1993): "Mobile Open Systems Technologies for the Utilities Industries", *Presented at the IEE Colloquium on CSCW Issues for Mobile and Remote Workers*, London, U.K., March 1993.
- [Davies,93b] N. Davies, G.S. Blair, A. Friday, A.D. Cross, P.F. Raven: Mobile Open Systems Technologies for the Utilities Industries, *Presented at the IEE Colloquium on CSCW Issues for Mobile and Remote Workers*, London, U.K., 16 March 1993.

- [DME,91] Open Software Foundation: "Distributed Management Environment Rationale", Open Software Foundation, October 1991.
- [Ellis,89] Ellis, C.A. and Gibbs, S.J.: "Concurrency Control in Groupware Systems", *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, Portland, Oregon, ACM Press, pp 399-407, 1989.
- [Fish,88] Fish, R.S., Kraut, R.E., Leland, M.D. and Cohen, M.: "Quilt: A Collaborative Tool for Cooperative Writing", *Published in Allen R.B. (ed), "Proceedings of the Conference on Office Information Systems (COIS'88)"*, March 1988, Palo Alto, California.
- [Gall,91] Gall, D.L., "MPEG: A Video Compression Standard for Multimedia Applications", *Communications of the ACM*, Vol 34, No 4, April 1991.
- [Garfinkel, 89] Garfinkel, D., Gust, P., Lemon, M. and Lowder, S.: "The SharedX Multi-user Interface User's Guide, Version 2.0.", Report STL-TM-89-07, Hewlett-Packard Laboratories, 3500 Deer Creek Road, Palo Alto, CA 94304.
- [Garret,86] Garret, L.N., Smith, K. and Meyrowitz, N.: "Intermedia: Issues, Strategies and Tactics in the Design of a Hypermedia Document System", *Proceedings of CSCW'86*, Austin, Texas, pp 163-175, December 1986.
- [Haake,92] Haake, J.M. and Wilson, B.: "Supporting Collaborative Writing of Hyperdocuments in SEPIA", *Proceedings of CSCW'92*, Toronto, November 1992.
- [Hughes,92] Hughes, J.A., Randall, D. and Shapiro, D.: "Faltering from Ethnography to Design", *Proceeding of CSCW'92*, Toronto, November 1992.
- [IAB,94] Internet Activities Board, RFC Documents, DDN Information Centre, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025, United States.
- [IEEE,94] IEEE Computer Society, The Standards Department, 1730 Massachusetts Avenue NW, 20036-1903 Washington DC, United States.
- [ISO7498,1984] International Organisation for Standardisation: Information Processing Systems - Open Systems Interconnection - Basic Reference Model, *Draft International Standard 7498*, October 1984.
- [ISO,92] ISO. Draft Recommendation X.901: Basic Reference Model of Open Distributed Processing - Part1: Overview and Guide to Use. *Draft Report ISO WG7 Committee*, 1992.
- [ISO,94] International Standards Organisation (ISO), Case Postale 56, CH-1211 Geneva 20, Switzerland.
- [Kreifelts,91] Kreifelts, T., Hinrichs, E., Klien K., Seuffert, P. and Woetzel, G.: "Experiences with the DOMINO Office Procedure System", *Proceedings of ECSCW 91*, Amsterdam, September 1991, Kluwer Academic Press, pp 117-131.
- [Levy,90] Levy, E. and Silberschatz, A.: "Distributed File Systems: Concepts and Examples", *Published in ACM Computing Surveys*, Vol. 22, No. 4, December 1990.
- [Lia,88] Lia, K.-Y. and Malone, T.W.: "Object Lens: A 'Spreadsheet' for Cooperative Work", *Proceedings of the Conference on Computer-Supported Cooperative Work (CSCW '88)*, ACM Press, pp 115-124, 1988.
- [Liou,91] Liou, M., "Overview of the px64 Kbps Video Coding Standard", *Communications of the ACM*, Vol 34, No 4, April 1991.
- [Medina-Mora,92] Medina-Mora, R., Winograd, T., Flores, R. and Flores, F.: "The Action Workflow Approach to Workflow Management Technology", *Proceedings of CSCW'92*, Toronto, November 1992.
- [Neufeld,92] Neufeld, G. and Vuong, S.: "An Overview of ASN.1", *Published in*

Computer Networks and ISDN Systems, Vol. 23, 1992, Pages 393-415.

[OMG,91] Object Management Group: The Common Object Request Broker Architecture and Specification. OMG Document Number 91.12.1, Revision 1.1, Draft 10, December 1991.

[O'Reilly,93] O'Reilly, J.J., Lane, P.M., Capstick, M.H., Salgado, H.S., Heidemann, R., Hofstetter, R. and Schmuck, H.: "MODAL: An Enabling Technology for Wireless Access", 4th IEE Conference on Telecommunications, Manchester, U.K, 1993.

[OSF,92] Open Software Foundation: OSF DCE Application Development Guide. Rev1.0 Update 1.01, 1992.

[OSF,92a] Open Software Foundation: OSF DCE Administration Guide. Rev1.0 Update 1.01, 1992.

[Reed,79] Reed, D.P. & Kanodia, R.K.: "Synchronisation with eventcounts and sequencers", *Communications of the ACM*, 22(2), pp115 - 123, February 1979.

[Robinson,91] Robinson, M. "Computer Supported Cooperative Work: Cases and Concepts" *Proceedings of Groupware'91*, Software Engineering Research Centre, Postbus 424, 3500 AK Utrecht, Nederland, pp 59-74, 1991.

[Rudkin,93] Rudkin, S. "Templates, types and classes in open distributed processing", *B.T. Technology Journal*, Volume 11, No3, pp32-40, June 1993.

[Satyanarayanan,93] M. Satyanarayanan, J.J. Kistler, L.B. Mummert, M.R. Ebling, P. Kumar and Q. Lu: Experience with Disconnected Operation in a Mobile Computing Environment, *Proc. of Mobile and Location-Independent Computing Symposium*, USENIX Association, 1993.

[Schroeder,93] Schroeder, M.D.: "A State-of-the-Art Distributed System: Computing with BOB", *Distributed Systems (2nd Edition)*, ACM Press, Edited by Sape Mullender, Addison Wesley, 1993.

[Shirley,92] Shirley, J.: "Guide to Writing DCE Applications", O'Reilly & Associates Inc., ISBN 1-56592-004-X.

[Shrivastava,91] Shrivastava, S.K., Dixon, G.N. and Parrington, G.D., "An Overview of the Arjuna Distributed Programming System", *IEEE Software*, January 1991.

[Stefik,87] Stefik, M., Foster, G., Bobrow, D., Kahn, K., Lanning, S. and Suchman, L.: "Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings", *Communications of the ACM*, 30(1): pp 32-47, 1987.

[Trigg, 86] Trigg, R., Suchman, L. and Halasz, F.: "Supporting Collaboration in Notecards", *Proceedings of CSCW'86*, Austin, Texas, December 1986.

[X/Open,94] X/Open Company Ltd., Apex Plaza, Forbury Road, Reading, Berks.,RG1 1AX, United Kingdom.