A SEMI-FORMAL COMPARISON BETWEEN THE COMMON OBJECT REQUEST BROKER ARCHITECTURE (CORBA) AND THE DISTRIBUTED COMPONENT OBJECT MODEL (DCOM)

by

PIETER WYNAND CONRADIE

submitted in part fulfilment of the requirements
for the degree of

MASTER OF SCIENCE

in the subject

COMPUTER SCIENCE

at the

UNIVERSITY OF SOUTH AFRICA

SUPERVISOR : PROFESSOR C H BORNMAN

JUNE 2000

Student number : **783-342-3**

I declare that **A SEMI-FORMAL COMPARISON BETWEEN THE COMMON OBJECT REQUEST BROKER ARCHITECTURE (CORBA) AND THE DISTRIBUTED COMPONENT OBJECT MODEL (DCOM)** is my own work and that all the sources that I have used or quoted have been indicated and acknowledge by means of complete reference.

P Conradie
SIGNATURE
(MR P W CONRADIE)

1 AUGUST 2000
DATE

Preface

The way in which application systems and software are built has changed dramatically over the past few years. This is mainly due to advances in hardware technology, programming languages, as well as the requirement to build better software application systems in less time. The importance of mondial (world-wide) communication between systems is also growing exponentially. People are using network-based applications daily, communicating not only locally, but also globally. The Internet, the global network, therefore plays a significant role in the development of new software. **Distributed object computing** is one of the computing paradigms that promise to solve the need to develop client/server application systems, communicating over heterogeneous environments.

This study, of limited scope, concentrates on one crucial element without which distributed object computing cannot be implemented. This element is the communication software, also called middleware, which allows objects situated on different hardware platforms to communicate over a network. Two of the most important middleware standards for distributed object computing today are the **Common Object Request Broker Architecture (CORBA)** from the **Object Management Group**, and the **Distributed Component Object Model (DCOM)** from **Microsoft Corporation**. Each of these standards is implemented in commercially available products, allowing distributed objects to communicate over heterogeneous networks.

In studying each of the middleware standards, a formal way of comparing CORBA and DCOM is presented, namely meta-modelling. For each of these two distributed object infrastructures (middleware), meta-models are constructed. Based on this uniform and unbiased approach, a comparison of the two distributed object infrastructures is then performed. The results are given as a set of tables in which the differences and similarities of each distributed object infrastructure are exhibited. By adopting this approach, errors caused by misunderstanding or misinterpretation are minimised. Consequently, an accurate and unbiased comparison between CORBA and DCOM is made possible, which constitutes the main aim of this dissertation.

Table of Contents

List of Figures

List of Tables

List of Appendixes

Appendix A

Appendix B

Appendix C

List of Acronyms

2PC	Two-Phase Commit protocol
ACID	Atomicity, Consistency, Isolation, and Durability
AFS	Andrew File System
API	Application Program Interface
BOA	Basic Object Adapter
CDS	Cell Directory Service
CIOP	Common InterOperability Protocol
CLSID	ClaSs IDentifier
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
COSS	Common Object Service Specification
DCE	Distributed Computing Environment
DCOM	Distributed Component Object Model
DEC	Digital Equipment Corporation
DFS	Distributed File Service
DII	Dynamic Invocation Interface
DNS	Domain Name Service
DSI	Dynamic Skeleton Interface
DSOM	Distributed System Object Model
DTC	Distributed Transaction Co-ordinator
DTS	Distributed Time Service
DTP	Distributed Transaction Processing
DLL	Dynamic Link Library
ECMA	European Computer Manufacturers Association
EER	Extended Entity Relationship
ESIOP	Environment Specific Inter-Orb Protocol
EXE	EXEcutable
GDA	Global Directory Agent
GDS	Global Directory Service
GIOP	General Inter-ORB Protocol
HP	Hewlett Packard
IBM	International Business Machines
IDL	Interface Definition Language

IID	Interface IDentifier
IIOP	Internet Inter-ORB protocol
ISO	International Standards Organisation
IOR	Interoperable Object References
IP	Internet Protocol
GUID	Globally Unique IDentifier
LRPC	Lightweight Remote Procedure Call
MIDL	Microsoft Interface Definition Language
MOM	Message Oriented Middleware
MSMQ	MicroSoft Message Queue
MTS	Microsoft Transaction Server
ODBC	Open DataBase Connectivity
ODMG	Object Database Management Group
OID	Object IDentifier
OLE	Object Linking and Embedding
OLE DB	OLE DataBase
OMA	Object Management Architecture
OMG	Object Management Group
OO	Object-Orientation, Object-Oriented
OODB	Object-Oriented Database
OOM	Object-Oriented Middleware
ORB	Object Request Broker
ORPC	Object Remote Procedure Call
OS	Operating System
OSF	Open Software Foundation
RFP	Request for Proposal
RPC	Remote Procedure Call
SCM	Service Control Manager
SQL	Structured Query Language
TCP	Transmission Control Protocol
TP	Transaction Processing
TSD	Task Structure Diagram
UUID	Universal Unique IDentifier
XDS	X/Open Directory Service
WWW	World Wide Web

Chapter 1 : Introduction

1.1 Overview

Because of the explosion of the Internet and the **World Wide Web (WWW)** in the last couple of years, most organisations are now using or planning to implement distributed object computing. Orfali, Harkey and Edwards (1996b) define **distributed object computing** as *'a computing paradigm that allows objects to be distributed across heterogeneous environments, allowing each of the distributed objects (components) to dynamically assume the roles of clients and servers.'* This implies that objects can be implemented on different hardware platforms, executing various **Operating Systems (OSs)**, and communicating over dissimilar networks. Objects can also assume the roles of clients, using the services of other distributed objects, or servers, providing services to other distributed objects.

This study of limited scope concentrates on a specific research area of distributed object computing, namely the *communication mechanisms* of distributed objects over networks. These communication mechanisms, also known as **distributed object infrastructures** or **middleware**, provide distributed objects with the ability to communicate over heterogeneous environments. The two main distributed object infrastructures currently dominating this area are the **Common Object Request Broker Architecture (CORBA)** and **Distributed Component Object Model (DCOM)**. The intent of this research is therefore to analyse and formally compare CORBA and DCOM.

In this chapter, an overview of the dissertation is given. Firstly, background to the study is provided, outlining the motivation and the scope of the research. Then specific terminology used in the dissertation is given, followed by an overview of articles and research papers published in this specific research field. Lastly, the structure of the dissertation is highlighted, briefly describing the contents of each chapter.

1.2 Background and Scope

Distributed object computing is the latest innovation in software development, and heralds a new era of computing. Combining the power of **object-orientation** and **client/server computing**, it promises to have a profound impact on how software is developed and maintained. Previously, object-orientation was limited to one language and one hardware platform. Developers could therefore only create and reuse objects on the same computer. With the advent of client/server computing, communication between objects on different computers became possible. This development, plus the global success of the Internet, is the factors driving distributed object computing into the mainstream of software research and development.

Two primary objectives of distributed object computing is the simplification of system development and maintenance by creating *objects*, and to allow objects installed on different computers (distributed objects) to communicate over a network. To accomplish the last objective, distributed object computing implements *distributed object infrastructures*. Orfali *et al.* (1996b) defines **distributed object infrastructures**, also referred to as **middleware**, as *'applications that let distributed objects, also called components, interact across heterogeneous environments.'* Two prominent distributed object infrastructure standards currently available are CORBA from the Object Management Group and DCOM from Microsoft Corporation:

- CORBA enables the invocation of methods implemented by distributed objects over dissimilar networks. A CORBA implementation employs **Object Request Brokers (ORBs)**, located on both the client and the server, to create and manage client/server communications between objects. The ORB on the client side allows an object to make a request to an object on the server side without any prior knowledge of where the object resides, what language it is coded in, or what operating system it is running on.

- DCOM was unveiled in 1996 as Microsoft Corporation's solution to distributed object architectures, and is the CORBA standard's biggest competitor. DCOM, previously known as Network OLE, is an extension of the **Component Object Model (COM)**, adding network communication support to COM. Although DCOM possesses its own core network protocol (**Object Remote Procedure Call (ORPC)**), and has major architectural differences from CORBA, it successfully duplicates the capabilities of CORBA within Microsoft Corporation's operating systems.

These two middleware standards, implemented as distributed object infrastructure products, are crucial for the success of distributed object computing. They enable the actual **distribution** in distributed object computing, by allowing objects residing on different computers to communicate over dissimilar networks. By comparing the two middleware standards, researchers and information system professionals can identify the significant similarities and differences between CORBA and DCOM. Furthermore, the result is a valuable resource for organisations that are planning to implement distributed object computing. This dissertation's main objective is therefore to analyse and formally compare CORBA and DCOM by using meta-modelling, and not to attempt to identify a winner. Both CORBA and DCOM have not yet reached their mature state, increasing the possibility of improvement and conversion. Any conclusion at this stage would therefore soon become inaccurate and invalid.

Meta-modelling is generally considered as a means of formalisation, whereby a system or systems are conceptualised by utilising meta-models. Stam (1995) defines a meta-model as '*a conceptual model of a modelling system.*' In this study, the system being conceptualised consists of the two distributed object infrastructures, specifically three core elements of each, namely the **object model**, **architecture**, and **object services**. By conceptualising these specific elements of CORBA and DCOM using meta-models, a uniform and unbiased comparison is possible. The result is then given in a set of tables in which the similarities and differences of CORBA and DCOM are exhibited. It is important to note that meta-modelling

can serve many purposes, including the explicit and concise description of a system, the comparison of systems, and the formulation of a system. In this dissertation, meta-modelling is however utilised only for the purpose of comparing CORBA and DCOM conceptually. In the next section, specific terminology used in the study is elucidated.

1.4 Terminology

For correct interpretation of the concepts used in this dissertation, a short glossary of terms is given below:

An **object** is an identifiable and encapsulated entity, which provides one or more services that can be requested by a client.

Middleware is the software that allows the elements of applications to communicate over networks, despite underlying differences in protocols and operating systems.

Object-Oriented Middleware (OOM) is a specific category of middleware that manages the communication between objects over heterogeneous environments.

A **client object** (client) is the object that requests certain services from a server object.

A **server object** (server) is the object that provides certain services to client objects.

Client/server computing divides software applications into client and server modules, that can be executed on multiple distributed hardware platforms connected by means of a single or multiple networks.

Object Request Broker provides the mechanisms by which client objects make requests to server objects, and receive responses from server objects.

A **client stub (proxy)** defines how client objects invoke operations on server objects.

A **server stub (skeleton)** enables client objects to invoke operations on server objects, essentially doing the reverse of the client stub.

An **interface** to an object consists of definitions for supported operations, parameters that are passed, and possible return values.

Interface Definition Languages (IDLs) are used to define interfaces to objects.

Meta-modelling is defined in Stam (1995) as '*the process of the conceptualisation of a modelling system.*'

A **meta-process model** describes on a conceptual level the steps or activities of a method or process.

A **meta-data model**, also on a conceptual level, describes the concepts provided by the system or method.

In the next section, a selection of key articles and papers published on the comparison between CORBA and DCOM are discussed. This provides an overview on how this dissertation of limited scope supplements the research already performed.

1.5 Related Research

Various articles and papers have been published which informally compare CORBA and DCOM. The most prominent of these include Emerald Chung, Huang, Yajnik, Liang, Shih, & Wang (1997), Tallman & Bradford Kain (1998), Gopalan (1998), Carr (1997), Watson, Soley & Bradley (1997), Session, Vogel & Kim (1998), and Montgomery (1997). In this section, a short overview of each paper is given, concentrating on the elements and concepts used in the comparison, and the purpose served by each paper.

The paper by Emerald Chung *et al.* (1997) makes an *architectural comparison* between DCOM and CORBA, based on three distinct layers, namely the *basic programming layer*, *remote layer*, and *wire protocol layer*. The **basic programming layer** is what is visible to the developers of client and object server programs, using CORBA and DCOM. The **remote layer** makes communication possible across different processes on the same platform. The **wire protocol layer** further extends the *remote layer* by enabling communication between different platforms. This paper is of interest mainly to persons already familiar with either CORBA or DCOM.

The paper of Tallman & Bradford Kain (1998) examines CORBA and DCOM by defining a *decision framework* for selecting a distributed object infrastructure. Five criteria are used in the decision framework, namely *specification*, *object model*, *services*, *platform* and *tool support*, and *maturity*. **Specification** compares the CORBA and DCOM specification, discussing its formalisms, usability, and conciseness. The second criterium, the **object model**, outlines each distributed object infrastructure's support for OO principles. **Services** define the basic CORBA and DCOM services, comparing their inherent resemblance. The second last criterium, **platform and tool support**, outlines CORBA and DCOM support on desktop and server platforms, plus the range of development tools available, including design tools, compilers, debuggers, and performance and testing tools. The last criterium, **maturity**, highlights the maturity of CORBA and DCOM by means of their use in software development projects. The main intent of this paper is to enable an organisation or individual to select the appropriate infrastructure for use in a distributed project.

Another key paper comparing CORBA and DCOM is that of Gopalan (1998), which examines the differences from a programming and architectural viewpoint. In this comparison a stock market example is used. A method called *get_price()* is coded to get the stock value of a particular stock, utilising CORBA and DCOM. This paper therefore concentrates on the programming aspect of each distributed object infrastructure, and can be of benefit to developers starting out on distributed object computing. The article by Carr (1997) gives an overview of CORBA and DCOM in terms of the protocols, maturity, availability, supporters, and interaction with the Internet. It is interesting to note that Netscape and Oracle are CORBA's most outspoken supporters, while Microsoft Corporation only endorses DCOM.

In the paper by Watson *et al.* (1997), a summarised comparison between ActiveX, another Microsoft defined name for DCOM, and CORBA is given. This comparison is based on seven distinct criteria, namely *specification*, *cross-platform support*, *cross-language support*, *maturity*, *Internet support*, *security*, and *scalability*. For the first criterium, namely **specification**, specific attention is drawn to the fact that the DCOM specification is not really open, but still under Microsoft control. **Cross-platform support** highlights the fact that DCOM is generally available only for Microsoft operating systems, while CORBA has consciously avoided dependencies on any particular operating system. The third criterium, **cross-language support**, outlines the fact that the programming model on which DCOM is based, is closely related to C++, while CORBA implements a more language-neutral approach, accommodating a wide range of programming languages. **Maturity** discusses the relatively new DCOM specification, against the older CORBA specification. **Internet support** states that ActiveX (DCOM) is based on native machine code, requiring a separate version for every ActiveX component (distributed object), while CORBA uses the programming language Java, enabling full cross-platform support. The second last criterium, **security**, outlines each infrastructure's security architecture. The final criterium, **scalability**, highlights the fact that from the start CORBA was designed with Internet-scale applications in mind, while DCOM was not. This paper, published by the OMG, may however be viewed as being more supportive of CORBA.

In the last two articles to be discussed here, namely Session *et al.* (1998) and Montgomery (1997), the differences between DCOM and CORBA are further explored. Traditionally, CORBA was used for non-Microsoft OSs, and DCOM for Microsoft operating systems. Microsoft however turned the management of DCOM over to the Open Group, who introduced DCOM to many non-Microsoft operating systems like UNIX. Choosing between CORBA and DCOM has therefore become extremely difficult, not only because of the many technical and strategic elements that must be considered, but also because both infrastructures are constantly changing.

Although all of the above papers and research articles are of great benefit for distributed object computing, from a functional point of view, they lack one fundamental aspect, namely a conceptual basis. All are informal, utilising ad-hoc comparative criteria. Many are also published by organisations favouring a specific infrastructure, for example the paper by Watson *et al.* (1997), published by the OMG. The need for an

accurate and *unbiased conceptual comparison* between CORBA and DCOM is therefore apparent. In the following section, the structure of the study is discussed.

1.6 Structure of Study

This comparative study of CORBA and DCOM consists of six chapters. Since CORBA and DCOM are viewed as middleware, specifically **object-oriented middleware**, the next chapter outlines middleware and the main middleware categories. CORBA and DCOM are then expounded in chapters 3 and 4 respectively, discussing each standard in detail. The actual comparison between CORBA and DCOM is then considered in chapter 5, analysing and comparing the two distributed object infrastructures, using a meta-modelling approach. The final chapter provides a conclusion to the study, summarising the research and results obtained.

The study can therefore be divided as follows:

Chapter 1: Sets the background of the comparative study, and gives an overview of the structure of the dissertation.

Chapter 2: Describes middleware in more detail, concentrating on its main categories.

Chapter 3: Gives an overview of CORBA.

Chapter 4: Gives an overview of DCOM.

Chapter 5: Compares CORBA and DCOM using meta-modelling.

Chapter 6: Concludes this comparative study.

Appendix A: Meta-models of CORBA.

Appendix B: Meta-models of DCOM.

Appendix C: Compares the meta-process models of the DCOM infrastructure with the meta-process models of the CORBA infrastructure.

1.7 Summary

In this chapter, the comparative study of the two major distributed object infrastructure standards is introduced. Firstly, distributed object computing and the two distributed object infrastructure standards are

briefly discussed, providing background to the dissertation and its scope. Then specific terminology used in the dissertation is listed, including *object*, *middleware*, *object-oriented middleware*, *client*, *server*, *client/server computing*, *object request broker*, *proxy*, *skeleton*, *interface*, *interface definition language*, *meta-modelling*, *meta-data model* and *meta-process model*. This is followed by an overview of key articles and papers published in this specific research field. In conclusion, the structure of the dissertation is highlighted, describing the contents of each chapter. In the next chapter, middleware, which forms such an important element of distributed computing, is discussed.

Chapter 2 : Middleware

2.1 Introduction

The previous chapter provides a short introduction to the comparative study of CORBA and DCOM, and distributed object computing. In this chapter, middleware is introduced. There are currently as many definitions of middleware as there are opinions about how applications need to be distributed. Some definitions refer to middleware as the software that allows the elements of applications to communicate over networks, despite underlying differences in protocols and operating systems (Douglas & Loosley 1997). Others view middleware as the essential component that provides the solution to application scalability, by creating a layer of distributed infrastructure, on top of which applications are deployed. However, everyone agrees that middleware is an essential component to accomplish distributed computing.

Distributed applications will typically be part of an environment that consists of heterogeneous operating systems, hardware platforms, communication protocols, and databases. By now it is universally known that for application developers, dealing with heterogeneous environments such as these, is becoming a nightmare. Middleware provides an isolation layer of software that shields application developers from this detail by presenting its own enabling layer. This layer hides the differences incurred by a heterogeneous environment. In effect, this layer disconnects applications from any dependencies on specific platforms. Middleware can therefore be considered as *'the enabling layer of software that resides between the application and the networked layer of heterogeneous platforms and protocols'* (Orfali, Harkey & Edwards 1996a).

In this chapter, client/server computing (*a form of distributed computing*), middleware, and the different categories of middleware are highlighted. Serving mainly as background for subsequent chapters, it also classifies CORBA and DCOM into a specific middleware category, namely **Object-Oriented Middleware (OOM)**.

2.2 Client/Server Computing

Douglas and Loosley (1997:712) defines **client/server computing** as *'a form of distributed computing in which an application is divided, or partitioned, into discrete processes, each of which typically executes on a different computing platform appropriate to the specific requirement of that process.'* Stripped to its essentials, client/server computing can be viewed as an extension of the fundamental idea of modular programming. *Modular programming* separates large software applications into modules, allowing for easier

development, and improved maintainability. Client/server computing however extends the concept of modular programming to that of a distributed environment, by recognising that those modules do not have to be executed on a single hardware platform. Instead, *multiple distributed hardware platforms* can be used, connected by means of different networks.

2.2.1 Client and Server

In client/server computing, the term client/server refers to a relationship between two processes, namely the client and server (Orfali *et al.* 1996a). The client is the process that requests work to be done on its behalf by the server process. The Oxford Dictionary of Computing (1997:76) defines a **client** as *'a system or process receiving a service.'* In most situations, whichever is the client or the server, is determined by the relationship of requester (client) to server. Servers provide services to requesting clients. The Oxford Dictionary of Computing (1997:444) defines a **server** as *'a system or process that provides services to clients.'* By definition, client/server computing is distributed. This implies that client and server processes normally execute separately on different hardware platforms, connected by a network. To facilitate the communication between client and server processes, middleware is necessary. This is discussed further in the next section.

2.3 Middleware

In (Orfali *et al.* 1996a), **middleware** is defined as *'the vague term that covers all the distributed software needed to support interactions between clients and servers.'* Douglas and Loosley (1997:712) define **middleware** as *'the enabling layer through which clients and servers processes, implemented on heterogeneous hardware platforms, communicate over networks, based on heterogeneous communication protocols.'* Middleware, implemented on all client and server platforms, is therefore by definition responsible for the transport of all requests and responses between client and server processes. Consequently, the use of middleware can be viewed as essential in the implementation of client/server computing, providing the enabling layer between the application and the network.

The main advantage of middleware is that it provides standardised **Application Program Interfaces (APIs)**, through which communication services are provided to client and server processes, instead of low-level network protocol primitives. Rauch (1996:193) defines APIs as *'high-level programming interfaces between application programs and various types of system software'*. In many respects middleware, also called the *client/server infrastructure*, is the glue that holds client/server applications together, as well as the productivity tool that saves developers from having to attend to low-level network primitives. It is a layer that exists between the application and the underlying complexities of the network. Put simply, middleware places an easy-to-use API between the application and the network.

Five main categories of middleware can be identified (Orfali *et al.* 1996a). These include:

- **Remote Procedure Call (RPC)** based middleware,
- **Message Oriented Middleware (MOM)**,
- Database middleware,
- **Transaction Processing (TP)** monitors, and
- object-oriented middleware.

Each category is described in *sections 2.3.1* to *2.3.5*.

2.3.1 Remote Procedure Call (RPC) based middleware

Remote procedure call based middleware is one of the most mature middleware categories for building client/server applications (Orfali *et al.* 1996a), and extends the familiar programming model (the procedure call) across the network. Evolving in the 1980's from UNIX environments, RPCs are widely accepted as technology capable of building distributed applications. The most important RPC based middleware products are implementations of the **Open Software Foundation's (OSF) Distributed Computing Environment (DCE)** open source code standard. The Open Software Foundation, now called the Open Group, was formed in 1988 as a non-profit research and development organisation devoted to open software, that is, software with standardised and publicised interface specifications (Orfali *et al.* 1996a). One of its most important source code standards is the Distributed Computing Environment, which specifies specific services for distributed computing. The services, presented in *Figure 2.1*, include the **Thread Service**, **Remote Procedure Call**, **Directory Services**, **Security Service**, **Distributed Time Service (DTS)**, and **Distributed File System (DFS)**.

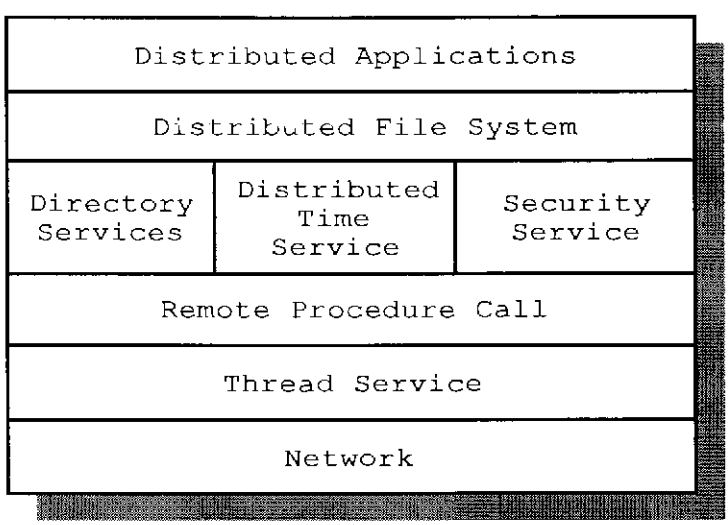


Figure 2.1: Services of the OSF's DCE

[from (Orfali *et al.* 1996a)]

Each service is now discussed in more detail:

- The **Threads Service** supports the creation and management of multiple threads of control within a client or server. This capability becomes particularly important within the context of an RPC. The RPC is synchronous by nature, implying that a client makes a call for a remote function and then waits until the call is fulfilled. With threads however, one thread can make the request, but another can begin to process the data from a different request. Threading can therefore greatly improve the performance of a distributed application.

- **Remote Procedure Call** is the mechanism by which clients invoke procedures in servers. A client may use directory services to bind to a particular server of interest at run time, and the client and server may use security services to guarantee desired levels of authentication, authorisation, integrity, and privacy.

- Finding system objects like users, organisations, groups, computers, printers, files, and processes in a distributed environment is the task of the **Directory Services**. Four distinct Directory Services can be identified, namely the **Cell Directory Service (CDS)**, **Global Directory Agent (GDA)**, **Global Directory Service (GDS)**, and the **X/Open Directory Service (XDS)**, outlined in *Table 2.1*.

Service	Description
Cell Directory Service (CDS)	A network cell is a group of systems administered as a single entity. The CDS is optimised for local access. The bulk of directory service queries enquire about resources within the query originator's cell. Each network cell needs at least one CDS.
Global Directory Agent (GDA)	The GDA is a naming gateway that connects the DCE domain to other administrative domains through the X.500 global directory service and **Domain Name Service (DNS)**.
Global Directory Service (GDS)	Based on the X.500 standard, the GDS functions as a higher level of the directory hierarchy in order to connect multiple cells in multiple organisations.
X/Open Directory Service (XDS)	Support for the X/Open API for directory service calls allows developers to write applications independent of the underlying directory service architecture. An XDS-compliant application will work unmodified with both DCE and X.500 directory services.

Table 2.1: The Four Directory Services

- There are two broad general categories of security services namely *authentication* and *authorisation*. Authentication verifies the identity of an entity, for example a user or a service. Authorisation or access control grants privileges to the entity, such as access to a file. The **Security Service** is based on the Kerberos authentication system, developed at MIT's Project Athena. Kerberos uses private-key encryption to provide three levels of protection. The lowest level requires only that user authenticity be

established at the initiation of a connection, assuming that subsequent network messages will flow from the authenticated principal. The next level up requires the authentication of each network message. On the level beyond these safe messages are private messages, where each message is encrypted as well as authenticated.

- Distributed network systems need a consistent time service. Many distributed services, such as distributed file systems and authentication services, compare dates generated on different computers. For the comparison to be meaningful, DCE must support a consistent time stamp. The **Distributed Time Service** is a system that provides time to other systems for the purpose of synchronisation.

- The **Distributed File System** provides transparent access to any file sitting on any node on the network. Based on the **Andrew File System (AFS)** from Transarc Corporation, it implements a single logical file system that is available through the Directory Services.

Almost all of the DCE objects are identifiable by **Universal Unique IDentifiers (UUIDs)**. A client that communicates to a server by means of RPC can therefore identify a specific resource by using a UUID. For example, a print server might generate object UUIDs for the different printers it controls, and a client submitting a print request would specify the desired printer UUID. Object type (class) UUIDs can also be generated, thereby associating different object UUIDs with one object type. For example, a print server might associate one object type UUID with RPC handlers that support line printers, and another object type UUID with a corresponding set of RPC handlers that support PostScript printers. UUIDs will be further discussed in *Chapter 4*.

Two of the more important elements implemented by all RPC based middleware products include **client/server stubs,** and an **Interface Definition Language (IDL)**, discussed in more detail in the following paragraphs.

(i) Client/Server Stubs

For client and server processes to communicate using RPCs, it is required that every server process that a client application can call, should be represented by a *placeholder* or *stub* (Linthicum 1997). This *client stub* looks like the server process to the application, and therefore provides *location transparency* of the service to the client. The client process simply calls the server process through the client stub as if it were a local process. The situation on the server side is similar. Here the *server stub* calls the server process, in the same manner as it would have been called by the client process, if it resided on the client platform.

It is important to note that most RPC products support synchronous communication between client and server processes. Synchronous communication implies that the initiating process sends a message or

request, and waits for a response before continuing (Douglas & Loosley 1997). *Figure 2.2* illustrates this synchronous calling procedure.

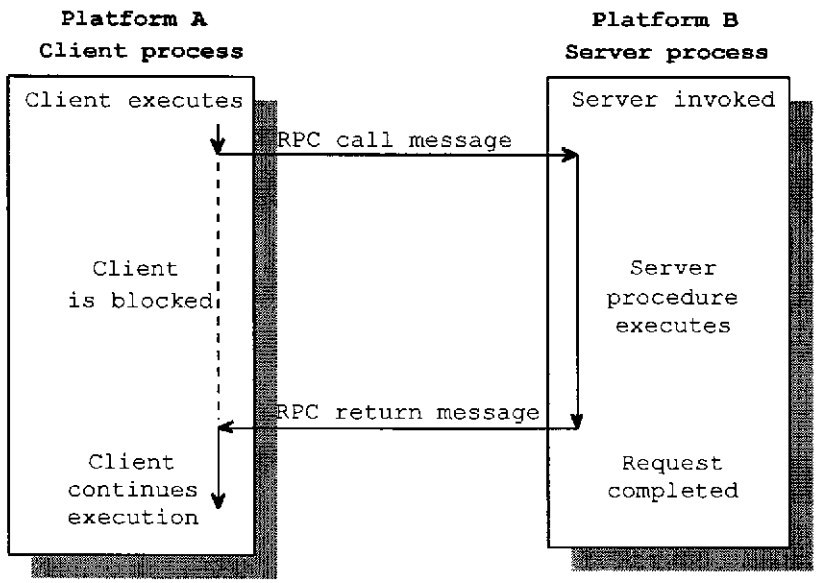


Figure 2.2: Remote Procedure Call (RPC) calling procedure

[from (Orfali *et al.* 1996a)]

When the client process on platform A calls a server process on platform B, the client process on platform A gets blocked. The server process on platform B then executes. On completion of the server process, a return message is received by the client process on platform A, enabling it to execute further.

(ii) Interface Definition Language (IDL)

A key component of RPC middleware is the translation of the representation of data, depending on the specific hardware platform and network protocol used. This process is commonly referred to as *marshalling* (Linthicum 1997). Marshalling requires a complete description of all data that are involved in a request including their types, format, and length, normally specified in an IDL. In (Orfali *et al.* 1996a), an IDL is defined as *'a high-level specification language, used to define interfaces that represent contracts between client and server applications.'* Interfaces normally consist of definitions for supported procedures, parameters that are passed, and possibly a return result. Once the IDL has been defined, it is used as input to an IDL compiler, which generates the client stub and server stub, enabling transparent communication between the client and server processes. While the IDL is a key element of RPC based middleware, it has also been re-discovered by object-oriented middleware. The most notable is the Object Management Group's IDL, which forms part of the CORBA standard, and Microsoft's IDL, used in their DCOM product.

2.3.2 Message Oriented Middleware (MOM)

Message oriented middleware refers to the process of distributing data and control through the exchange of records, known as messages (Rauch 1996). Messages are strings of bytes that are meaningful to the applications that exchange them. MOM is perhaps the most confusing middleware category today. There is no open source code standard as in the case of the OSF's DCE for RPC middleware, nor an open specification as in the case of CORBA. Most MOM products are based on one of three communication models, namely *message passing*, *message queuing*, and *publish/subscribe*.

- **Message passing** is a direct, program-to-program communication model. Using message passing, an application request is sent in the form of a message from one program directly to another. Both programs communicate with each other directly in a connection-oriented fashion. Message passing generally implies that the communication mechanism can either be *synchronous*, i.e. the sender is blocked until the receiver sends a message, or *asynchronous,* which implies that the initiating process sends a message or request, and continues processing without waiting for a response (Douglas & Loosley 1997). However, it is important to note that the message-passing model is always connection-oriented, meaning that a direct link between two programs that participate in this message exchange must be maintained.

- **Message queuing** is an indirect program-to-program communication model that allows programs on heterogeneous platforms to communicate via message queues, rather than by calling each other directly. Message queuing always implies a connectionless model. Therefore, the availability of the partner program is not mandatory. As illustrated in *Figure 2.3*, messages are placed in queues (queue A and B), which can be memory or disk based, for either immediate or subsequent delivery by the queue managers.

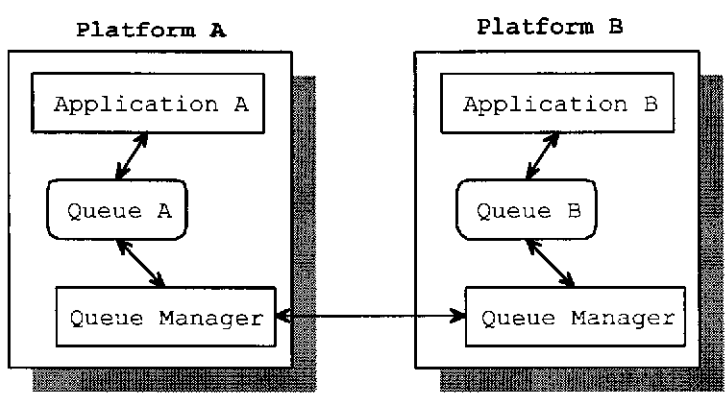


Figure 2.3: Message Queuing Model

[derived from (Rauch 1996:199) Figure 9.5)]

This allows programs to run independently, at different speeds, and without a logical connection between them.

- The **publish/subscribe** model has evolved from the real world of trading applications. Although this can still be considered as a niche technology represented by just a few products, the technology has already achieved a certain level of maturity. As illustrated in *Figure 2.4*, the publisher publishes information on the network via the message bus. This information is then available to subscribers, for example A, B, and C, which subscribe to the publisher.

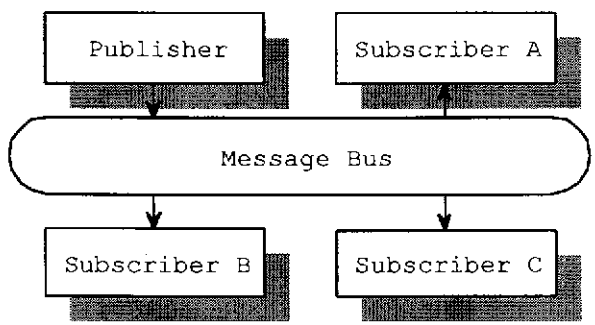


Figure 2.4: Publish/Subscribe Model

[derived from (Rauch 1996:201) Figure 9.6)]

These subscribers can then consume the particular information, utilising the message bus.

2.3.3 Database middleware

Database middleware is one of the largest middleware categories today (Orfali *et al.* 1996a). Mainly based on RPC middleware products, it provides universal and consistent data access to many data sources including relational databases, and **Object-Oriented DataBases (OODBs)**. Tkach and Puttick (1996) define an OODB as *'a storage manager for objects, allowing the transparent management of object persistence.'* Database middleware products are however vendor specific, not supporting heterogeneous hardware platforms or databases. This implies single-server, single-vendor applications.

All database middleware products provide support for **Structured Query Language (SQL)**, the standard database language used to access and manage data in a database. A desktop client requests data from a database through an SQL statement. The SQL statement is translated or compiled by the database server into native database code used to navigate the database, find the requested data, and send it to the client. SQL statements are commonly implemented by means of APIs. These APIs are based on de-facto standards, like the Microsoft Corporation **Open DataBase Connectivity (ODBC)** specification, or proprietary database vendor APIs.

Many database vendors have also implemented specific extensions to the SQL standard, for example simple database transactions, also called stored procedures. Orfali *et al.* (1996a:161) define **stored procedures** as *'named collections of SQL statements and application logic that is compiled, verified, and stored on database servers.'* When called, a stored procedure executes like a transaction, supporting the ACID properties outlined in *section 2.3.4(i)*. The main disadvantage of stored procedures is that it allows no synchronisation, each stored procedure executes separately without any interaction with other stored procedures. This allows stored procedures to implement only the flat transaction model, discussed in the next section.

RPC based middleware, like MOM and database middleware, are technologies that offer significant value to a wide variety of applications. However, they do not provide comprehensive support for synchronous transactions spanning heterogeneous hardware platforms and databases products.

2.3.4 Transaction Processing monitors (TP monitors)

Transaction Processing monitors is the fourth category of middleware. TP monitors specialise in managing transactions over several heterogeneous hardware platforms and databases, also known as resource managers. Linthicum (1997:124) defines a TP monitor as *'an operating system for transaction processing.'* TP monitors were developed from the ground up as operating systems for transactions. The basic unit of management, execution, and recovery is a transaction. With TP monitors, the application developers don't have to be concerned about issues like transaction concurrency, scheduling, failures, or load balancing. All is made transparent, very much like an operating system that makes the hardware transparent to the user. In the next few paragraphs the main elements and features of TP monitors are discussed, including *transactions*, *transaction models*, *funnelling*, *load balancing*, and *synchronisation*.

(i) Transactions

A transaction is defined in Orfali *et al.* (1996a:258) as *'a collection of actions which support the ACID properties'*. **ACID**, the abbreviation for **Atomicity, Consistency, Isolation, and Durability**, is outlined in *Table 2.2*.

Property	Description
Atomicity	Atomicity means that a transaction is an indivisible unit of work, all of its actions succeed or they all fail.
Consistency	Consistency implies that after a transaction executes, it must leave the system in a correct state or it must abort.
Isolation	Isolation implies that a transaction's behaviour is not affected by other transactions that execute concurrently.

Property	Description
Durability	Durability means that a transaction's effects are permanent after being committed.

Table 2.2: Description of ACID properties

(ii) Transaction models

It is important to note that different transaction models can be identified for transactions (Orfali *et al.* 1996a). The normal **flat transaction model**, as illustrated in *Figure 2.5*, has long been the workhorse of TP monitors. In this model all transactions are executed on one level, beginning with a *'begin transaction'*, and ending with either a *'commit transaction'* or *'abort transaction'*. *'Begin transaction'* indicates the start of a transaction, *'commit transaction'* indicates the successful end of a transaction, and *'abort transaction'* indicates the unsuccessful end of a transaction.

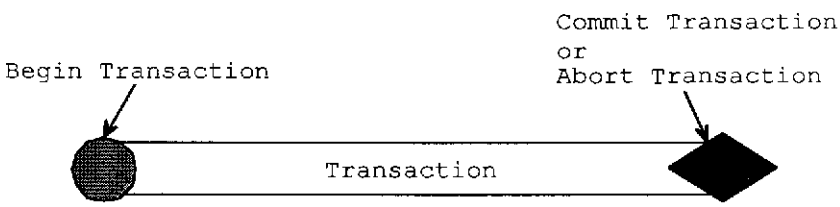


Figure 2.5: Flat Transaction Model

[derived from (Orfali *et al.* 1996a:260) Figure 16-1]

With newer transaction models such as the **nested transaction model**, illustrated in *Figure 2.6*, a transaction can be divided into sub-transactions, allowing each sub-transaction to execute on a different platform. Each sub-transaction can also start its own sub-transactions, making the entire structure recursive.

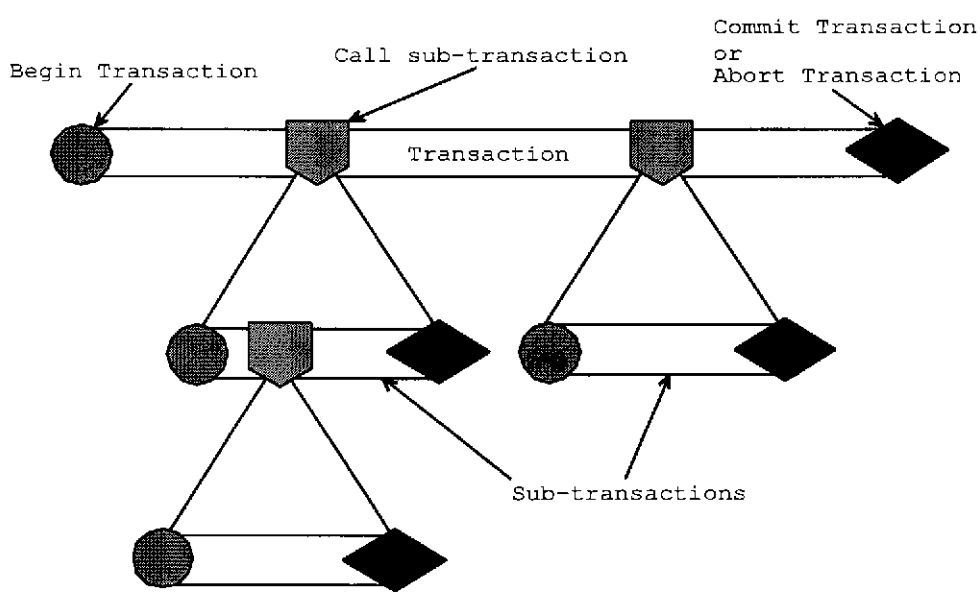


Figure 2.6: Nested Transaction Model

[derived from (Orfali *et al.* 1996a:274) Figure 16-13]

In the nested transaction model, a sub-transaction only becomes permanent after it issues a *'commit transaction'* and all its ancestors or parent transactions also issue commit transactions. If a parent transaction issues an *'abort-transaction'*, all descendant transactions will also issue abort transactions, thereby ensuring consistency between parent and sub-transactions.

(iii) Funnelling

One of the main advantages of TP Monitors is that it reduces the number of connections to host servers (Linthicum 1997). Without TP Monitors, each desktop client that connects to a host server, is represented by a separate server process. This implies that if 500 desktop clients are connected to a host server, 500 server processes must be started. A TP Monitor however acts as a connection concentrator, **funnelling** client connections into *process pools*. In Linthicum (1997:162), process pools are defined as *'shared groups of processes, pre-started by the TP monitor, waiting for work.'* Each process pool is linked by one or more connections to a server, thereby drastically reducing the amount of processes managed by the host server. This not only minimises host server overheads, but also increases the amount of desktop clients that can request services from one host server.

(iv) Load Balancing

As outlined previously, TP monitors remove the process-per-client requirement for resource managers by *funnelling* incoming client requests through process pools. When the number of incoming client requests surpasses the number of available process pools, the TP monitor automatically starts more process pools (Linthicum 1997). This feature is known as **load balancing**, which ensures that the workload is evenly balanced between client and server.

(v) Synchronisation

In client/server computing, updates to multiple resource managers have to be **synchronised** across all platforms on which these resource managers reside. One way to achieve such synchronisation is through the use of the **Two-Phase Commit (2PC)** protocol. The 2PC, divided into five distinctive steps, synchronises transactions across heterogeneous hardware platforms, and is part of the OSF's Transaction Processing standard published in 1992 (Orfali *et al.* 1996a).

The five steps are outlined in *Table 2.3*.

Step	Description
One	In the first phase of a commit, the designated transaction co-ordinator or commit manager node sends a *prepare-to-commit* command to all subordinate nodes that were directly asked to participate in the transaction.
Two	The first phase of the commit terminates when the transaction co-ordinator receives *ready-to-commit* signals from all its subordinate nodes that participate in the transaction.
Three	The second phase of the commit starts after the transaction co-ordinator makes the decision to commit the transaction, based on step two. The transaction co-ordinator then sends a *commit* command to all subordinate nodes.
Four	The second phase of the commit terminates when all subordinate nodes have committed, making their sub-transactions durable, and sending *completed* signals. The transaction co-ordinator then tells its client that the transaction has been completed.
Five	The two-phase commit aborts if any of the participants return a refuse signal, meaning that their part of the transaction or sub-transaction failed. If that happens, the transaction co-ordinator then sends an abort command to all subordinate nodes.

Table 2.3: Two-Phase Commit (2PC) Protocol

All major database vendors have implemented the 2PC protocol. There is however one constraint, transactional semantics (*'begin transaction', 'commit transaction', 'abort transaction'*) under 2PC can only be performed by databases from the same vendor. Two-phase commit protocol is therefore not supported if an application attempts to update heterogeneous databases, as would be the case for Informix and Sybase within one transaction.

TP Monitors are however capable of co-ordinating and managing transactions across heterogeneous databases. This is accomplished by implementing 2PC on TP monitors, and by TP monitors and databases supporting the XA-interface. The XA-interface is a standardised interface used for communication between resource managers and TP monitors, defined as part of the Open Group's X/Open **Distributed Transaction Processing (DTP)** specifications (Linthicum 1997). Adherence to this common protocol therefore allows TP Monitors to preserve the integrity of updates across heterogeneous databases.

2.3.5 Object-Oriented Middleware (OOM)

So far, various forms of middleware, all of them non-object-oriented, were discussed. However, many organisations have adopted, or are in the process of adopting, long term object-oriented strategies. They are therefore looking for middleware that provides a higher level of abstraction in order to match OO tools, databases, and programming languages. Such middleware is often referred to as object-oriented

middleware (Linthicum 1997). Based on OO concepts, namely *objects*, *classes*, *encapsulation*, *inheritance*, and *polymorphism*, as outlined in *Table 2.4*, it forms the most important enabling technology for distributed object computing.

Concept	Description
Object	An object is a representation of a real-world entity, for example person, or logical idea such as bank account. It consists of three key elements, namely *identity*, *state*, and *behaviour*. An object's ***identity*** is defined by its **Object IDentifier (OID)**, a unique and invariant attribute that distinguishes it from other objects. An object's ***state*** is the collection of all attribute values it possesses at a given time (Jones 1995). The ***behaviour*** of an object is defined by a set of operations or methods. These methods are defined in the class to which the object belongs.
Class	A class is a description, also called a template or blueprint, that defines the structure (attributes and methods) of objects. The concepts *object* and *class* are closely related. An object is an *instance* of a class, and a class is a logical grouping of objects having the same structure.
Encapsulation	Encapsulation is the concept of hiding data and operations by only providing interfaces to the operations that manipulate the data. Encapsulation is defined in Jones (1995:9) as *'the grouping of related ideas into one unit, which can be referred to by a single name.'*
Inheritance	Jones (1995:18) defines inheritance as 'the mechanism by which a class (subclass) may be specialised from more general classes (superclasses).' Two common types of inheritance are **single** and **multiple inheritance**. With **single inheritance**, a subclass may only inherit methods and attributes from a single superclass. With **multiple inheritance**, a subclass may inherit methods and attributes from more that one superclass.
Polymorphism	The word *polymorphism* comes from two Greek words namely *poly* meaning many, and *morphos* meaning form or shape. Something that is polymorphic has therefore the ability to take on many forms. Class-based polymorphism is defined in Jones (1995:34) as *'the facility by which a single method name may be defined upon more than one class, and may take on different implementations in each of those classes.'*

Table 2.4: The main Object-Oriented (OO) concepts

Object-Oriented Middleware can therefore be defined as *'the middleware that manages the communication between objects over heterogeneous operating systems, hardware platforms, and communication protocols* (Linthicum 1997).' Two popular OOM standards that are currently dominating the OOM market are CORBA and DCOM. The CORBA standard, discussed in *chapter 3*, has the biggest vendor support base with the Object Management Group consisting of more than 800 members. The Microsoft Corporation's DCOM standard, discussed in *chapter 4*, is relatively new and fully supported by Microsoft on all its operating systems.

2.4 Summary

In this chapter, a brief introduction to middleware is given, concentrating on client/server computing and the five main middleware categories. These five categories, namely RPC based middleware, message oriented middleware, database middleware, TP monitors, and object-oriented middleware, presents an synopsis of the current state of middleware. It is important to note that as the middleware industry continues to evolve, the five categories of middleware may also evolve. One belief is that consolidation rather than further fragmentation of middleware categories will occur in the future. Such consolidation might result from mergers and acquisitions of middleware companies, or from products belonging to one category of middleware, beginning to offer functionality that is today provided by another type of middleware product. This is especially true for TP Monitors and RPC based middleware being incorporated more and more into OOM. Regardless of what the future might hold, middleware is viewed as an essential element of any serious distributed application. In the next chapter, the first of the two main object-oriented middleware standards, namely CORBA, is introduced.

Chapter 3 : Common Object Request Broker Architecture (CORBA)

3.1 Introduction

In the previous chapter, the main middleware categories were outlined. In this chapter, the object-oriented middleware standard CORBA is discussed. CORBA is the standard from the **Object Management Group (OMG)** for implementing a distributed object infrastructure, based on the concept of an **Object Request Broker (ORB)**. As defined by the OMG in the Object Management Architecture Guide, the ORB *'provides the mechanisms by which objects transparently make requests and receive responses* (Baker 1997).' The *Object Management Architecture Guide* defines the OMG's technical objectives and terminology, and describes the conceptual models upon which OMG standards are based.

Bouzeghoub, Gardarin and Valduriez (1997:362) define an ORB as *'the middleware that enables a client object to send a message to a distant server object and to receive a response, without needing to know the location of the server.'* A CORBA compliant ORB is therefore the middleware that establishes the client/server relationship between objects located in a heterogeneous environment. Since CORBA forms only a **standard**, and not a **product**, it is the responsibility of CORBA middleware vendors to produce CORBA products. The most important of these products include **Distributed System Object Model (DSOM)** from **International Business Machines (IBM)**, **ORB Plus** from **Hewlett Packard (HP)**, **ObjectBroker** from **Digital**, and **Orbix** from **Iona** (Orfali, Harley & Edwards 1996b).

In the following sections, the main elements of the CORBA standard are discussed. These include the **Object Management Architecture (OMA)**, OMG Interface Definition Language, and the architecture of CORBA. The developer and main driving force behind CORBA, namely the Object Management Group, is introduced in the following section.

3.2 Object Management Group (OMG)

In April 1989, a group of vendors who believed in the benefits of object-oriented software development, formed an industry coalition in an attempt to structure the chaos in the object marketplace (Bouzeghoub *et al.* 1997). The name given to the coalition was the OMG, and the CORBA specification is arguably its most important specification. From the modest 8 starting members, including 3Com Corporation, American Airlines, Canon, Data General, HP, Philips Telecommunications, Sun Microsystems and Unisys Corporation, the OMG quickly grew to 80 members by 1991, 200 members by 1992, and 300 members by

1993. Today it is an international organisation supported by over 800 members, including information system vendors, software developers and users.

The CORBA specification, although created and maintained by the OMG, is published in collaboration with another standard organisation, namely the Open Group (Mowbray & Ruh 1997). Core parts of the CORBA specification, in particular the OMG Interface Definition Language, has also been accepted by the **European Computer Manufacturers Association (ECMA)**, and the **International Standards Organisation (ISO)**, as a formal standard for specifying *object interfaces*. Object interfaces, also called interfaces, is a list of the operations and attributes that an object provides (Baker 1997). CORBA can therefore be viewed as both a *formal* and *de facto* standard. *Formal* through the ECMA and ISO acceptance of OMG IDL, and *de facto*, through the widespread adoption and support of the CORBA specification by middleware vendors.

3.3 Object Management Architecture (OMA)

The OMA is the framework on which all specifications, produced by the OMG, are based (Baker 1997). Published in the *Object Management Architecture Guide*, it provides for two fundamental models, namely the **Core Object Model** and the **Reference Model**.

3.3.1 Core Object Model

An object model can be defined as *'the term that refers to the collection of concepts used to describe objects in a particular object-oriented language, specification (middleware, database, etc.), or analysis and design methodology* (H7 1997). The main objectives of the Core Object Model are *portability* and *interoperability* (Baker 1997). The most important aspect of **portability** is design portability. This refers to the knowledge of an object's interface, and the ability to create applications whose components do not rely on the existence or location of a particular object implementation. The Core Object Model does not define the syntax of interface descriptions, but describes the semantics of types and their relationships to one another. **Interoperability** refers to the ability to invoke operations on objects regardless of where they are located, which hardware platform they execute on, or what programming language they are implemented in. This is achieved by the ORB, which relies on the semantics of objects and operations described in the Core Object Model (CORBA 1995).

The Core Object Model consists of a number of key concepts, namely *objects*, *types*, *operations*, *interfaces*, *subtyping*, *non-object types*, and *exceptions*, outlined in the following paragraphs.

(i) Objects

Objects are used to represent entities, for example a person, ship, or document. Each object has a unique identity, represented by an *object reference*, which is '*a value that unambiguously identifies an object* (Baker 1997)', and which cannot change over time. Objects encapsulate both state and behaviour.

(ii) Types

Objects are instances of types, also called classes. Types group objects, which represent entities belonging to a group.

(iii) Operations

Operations implement *requests*. Requests specify what operation is to be performed and what parameters are to be passed to the operation invocation. Each operation has a *signature* that includes an operation name, a set of parameters, and a set of result types.

(iv) Interfaces

The set of operation **signatures** defined in a type is collectively called the type's *interface*. The result is that every object that is an instance of the type, assumes the interface defined within the type. It is important to note that the interface of a type consists not only of the operation signatures defined within the type, but also of signatures that are inherited from supertypes.

(v) Subtyping

Types may be related through a subtype/supertype relationship, allowing the inheritance of supertype properties. The supertype of all objects is an abstract type *Object*.

(vi) Non-object types

The Core Object Model recognises the existence of values that are not objects, for example integer and real. These are usually called non-objects, data types or values.

(vii) Exception

An exception is an indication that an operation request was not performed successfully. An exception may be accompanied by additional, exception-specific information.

Two other important concepts, namely *components* and *profiles*, are also used in relation to the Core Object Model (Baker 1997). A **component** is an extension to the Core Object Model that provides more concrete specialisation of the concepts defined in the model. The Core Object Model, together with one or more *components*, produces what is called a **profile**. The CORBA Object Model, discussed later in this chapter, is therefore considered a profile.

3.3.2 Reference Model

The OMA Reference Model is an architectural framework for the standardisation of interfaces used by applications (Baker 1997). Illustrated in *Figure 3.1*, it consists of five elements, namely the *Object Request Broker*, *Object Services (CORBAservices)*, *Common Facilities (CORBAfacilities)*, *Domain Interfaces (CORBAdomains)*, and *Application Objects*, each discussed in the following paragraphs (CORBA 1995).

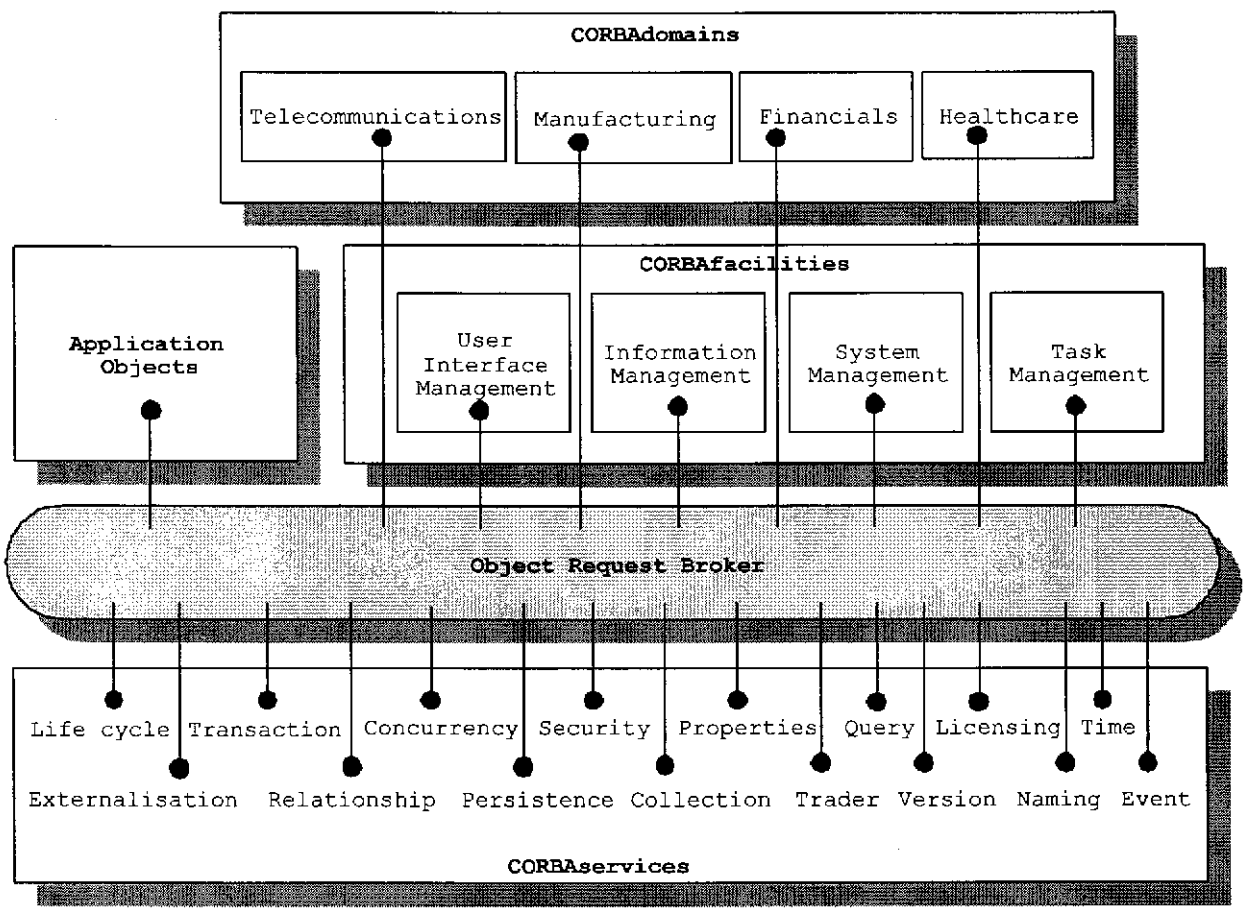


Figure 3.1: Object Management Architecture Reference Model

[(Baker 1997:11) Figure 1.4]

(i) Object Request Broker

The Object Request Broker element, also referred to commercially as CORBA, is the foundation for building applications from distributed objects, and for interoperability between these distributed objects over heterogeneous environments. The ORB enables objects to transparently make and receive requests and responses across heterogeneous networks, operating systems, and hardware platforms.

(ii) Object Services (CORBAservices)

The Object Services element, also called **CORBAservices**, is a collection of services that support basic functions for using and implementing objects (Bouzeghoub *et al.* 1997). Interfaces are provided to create objects, to control access to objects, to keep track of relocated objects, and other more sophisticated capabilities, such as query services, transaction services, concurrency control, and directory services.

(iii) Domain Interfaces (CORBAdomains)

Domain interfaces, also called **CORBAdomains**, represent vertical areas that provide functionality of direct interest to end-users in particular application *domains*. A domain can be defined as *'a formal boundary that defines a particular subject or area of interest'* (Baker 1997). Domain interfaces may combine some Common Facilities and Object Services, but are designed to perform particular tasks for users within a certain vertical market or industry. This is by far the larger set of facilities. As industry groups develop such facilities, the OMG integrates their efforts into the Domain Interface architecture, based on the OMG IDL. Examples of vertical markets include finances, telecommunications, and manufacturing.

(iv) Common Facilities (CORBAfacilities)

The Common Facilities element, also called **CORBAfacilities**, represent a collection of services that many applications may share, but which are not as fundamental as Object Services. Examples of CORBAfacilities include User Interface Management and System Management. All of the CORBAfacilities may be applied in multiple vertical domains, like financial and telecommunications.

(v) Application Objects

Application Objects are products of a single vendor or in-house development group (CORBA 1995). Each application object or business object implements its own application interfaces specific to the application. Because the OMG does not develop applications but only specifications, these interfaces are not standardised. If over time a set of general services emerge from a particular application domain, the OMG can decide to incorporate it in future standards. In later sections, the elements of the OMG Reference Model

are again discussed, concentrating on the specific services they provide. In the next section, the ORB element is enunciated.

3.4 Object Request Broker

The Object Request Broker enables objects to co-operate over heterogeneous environments (Orfali *et al.* 1996b). Implemented by different vendors, it provides the core communication facility for objects over networks and operating systems. The first CORBA version, namely CORBA 1.0, represents the merging of all proposals made by the members of the OMG in 1991, hence the inclusion of the word *'common'* in the name. Two versions, namely CORBA 1.1 and CORBA 1.2, with minor corrections, were issued in 1992 and 1993 respectively. The experience gained from CORBA implementations resulted in the CORBA 2.0 revision, published in August 1996. Three version then followed, namely CORBA 2.1 (August 1997), CORBA 2.2 (February 1998), and CORBA 2.3 (October 1998) incorporated a few updates, and basic support for communication between CORBA and DCOM. Currently, CORBA 3.0 is the newest version of CORBA, published in September 1999. However, since CORBA 2.x forms the core of all currently available CORBA products, this study concentrates on the CORBA 2.x standard(s), highlighting the additions of the CORBA 3.0 standard later in the chapter.

In the following sections, the CORBA Object Model, the CORBA architecture, the OMG's Interface Definition Language, and static and dynamic invocation in CORBA, are discussed.

3.4.1 CORBA Object Model

The CORBA Object Model is a **profile** that extends the Core Object Model with several components (Baker 1997). The most important of these include *objects*, *requests*, *interfaces*, *operations*, *attributes* and *exceptions*. Although some components were already discussed in *section 3.3.1*, a short overview of each component is provided in relation to CORBA.

(i) Objects

An object is an identifiable, encapsulated entity that provides one or more services that can be requested by a client. It is necessary to distinguish between an object's implementation and its reference. The former is the code that implements the object's operations, while the latter is the object's identity used by clients to invoke its operations. Object references are handles to objects, and are defined by **Interoperable Object References (IOR)** in CORBA, allowing object references to be passed across heterogeneous ORBs. A given object reference will always denote a single object, but several distinct object references may denote the same object.

(ii) Requests

A client requests services by issuing requests. A request, defined as an event, consists of an operation name, a target object, and zero or more parameters.

(iii) Operations

An operation has a *signature* consisting of an operation identifier or name, the type of the value returned by the operation, and a list of parameters. Each parameter consists of a name, type, and direction indicator.

Two styles of execution semantics are defined by the CORBA Object Model, described in *Table 3.1*, namely *at-most-once* and *best-effort*.

Execution Semantic	Description
At-most-once	If an operation request returns successfully, it is performed exactly once. If it returns an exception indication, it is performed *at-most-once*.
Best-effort	A best-effort operation is a request-only operation, requesting an operation without waiting for a result.

Table 3.1: The Two Styles of Execution Semantics supported in CORBA

(iv) Interfaces and Attributes

An *interface* is a description of a set of possible operations that a client may request of an object. An object satisfies an interface if it can be specified as the target object in each potential request described by the interface. An interface may have *attributes*. An attribute is logically equivalent to declaring a pair of accessor functions, one to retrieve the value of the attribute, and one to set the value of the attribute. An attribute may be read-only in which case only the retrieval accessor function is defined.

(v) Exceptions

An *exception* is an indication that an operation request was not performed successfully (CORBA 1995). Exceptions are defined as a specialised non-object type in the CORBA Object Model, containing optional fields for providing information on the causes of abnormal operation termination. In the following section, the main elements of the CORBA architecture are highlighted.

3.4.2 The CORBA Architecture

In *Figure 3.2*, the main elements of the CORBA architecture are presented, including the *client*, *object implementation*, ORB Core, *stub*, **Dynamic Invocation Interface (DII)**, *skeleton*, **Dynamic Skeleton Interface (DSI)**, *Object Adapter*, *ORB interface*, *interface repository*, and *implementation repository* (Orfali *et al.* 1996b).

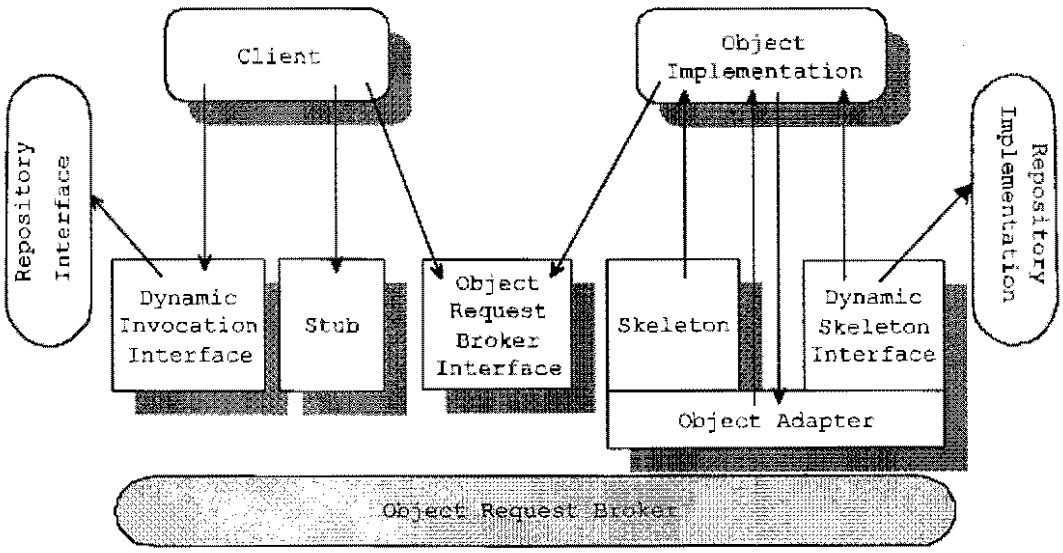


Figure 3.2: The main CORBA elements

[derived from (Orfali *et al.* 1996b:69) Figure 4.2]

The interaction of the elements of the CORBA architecture is explained later in the chapter, specifically when static and dynamic invocations in CORBA are discussed. The following paragraphs provide a short outline of each element.

(i) Client

Although a client is generally considered to be an entity initiating requests on another object, it is important to recognise that an entity is a client relative to a particular object (Orfali *et al.* 1997). To make a request, the client can either use **dynamic invocation interfaces** or **stubs**, discussed in the following paragraphs.

(ii) Object Implementation

The object implementation is the code and data that actually implements an object on a server, providing services to a requesting client. Requests for services are received as an up-call either through the **dynamic skeleton interface** or static generated **skeletons**.

(iii) ORB Core

The ORB Core is responsible for the communication of requests between clients and object implementations.

(iv) Stubs

The stubs, also called **client stubs**, provide the static interfaces to object operations. These precompiled stubs, generated by the IDL compiler, define how clients invoke corresponding operations of object implementations. From the client's perspective, the stub acts like a local call.

(v) Skeletons

The skeletons, also called **server stubs**, provide static interfaces to each operation exported by an object. Both client and server stubs are created by the IDL compiler.

(vi) Dynamic Invocation Interface (DII)

The *dynamic invocation interface* is an interface that allows the dynamic construction of object invocations. Therefore, rather than calling a stub routine that is specific to a particular operation on a particular object, a client may specify the object to be invoked, and the operation to be performed through a dynamic call.

(vii) Dynamic Skeleton Interface (DSI)

The *dynamic skeleton interface*, introduced in CORBA 2.0, is an interface that allows dynamic handling of object invocations. This means rather than being accessed through a skeleton that is specified to a particular operation, an object's implementation is reached through an interface that provides access to the operation name and parameters in a manner analogous to the client side's DII.

(viii) Object Adapters

Object adapters are specific interfaces for the purpose of creating interconnections between object implementations and the ORB Core (CORBA 1995). The main services offered by object adapters include the generation and interpretation of object references, invocation of operations, security checks, activation and deactivation of implementation of objects, associating object references with implementations, and keeping records of implementations.

Servers may support a variety of object adapters to satisfy different types of requests. For example, an object-oriented database may want to implicitly register all its fine-grained objects without issuing individual calls to the object adapter. In such a case, it doesn't make sense for an object adapter to maintain per-object state, so the OODB can provide a special-purpose object adapter that interfaces with the ORB and at the same time meets its own special requirements. The OMG however prefers not to see a proliferation of object adapters, and therefore specifies a **Basic Object Adapter (BOA)** in CORBA. This BOA must be supported in each ORB to be viewed as CORBA compliant.

(ix) ORB Interface

The ORB Interface is the direct interface to the ORB Core, and provides operations that are useful to both client and object implementations. These consist of a small number of operations manipulating object references, accessible both to clients and object implementations. For example, the ORB Interface enables the conversion of an object reference to a character string and vice versa.

(x) Interface Repository

The *interface repository*, introduced in CORBA 2.0, is the component of the ORB that provides persistent storage of interface definitions specified in IDL. This enables a program to find an object whose interface was not known when the program was compiled.

(xi) Implementation Repository

The *implementation repository* contains information that allows the ORB to locate and activate implementations of objects. All of these elements together allow distributed objects to communicate over heterogeneous environments.

3.4.3 OMG Interface Definition Language (OMG IDL)

The OMG interface definition language is used to describe the interfaces that client objects call and object implementations provide. Interface definitions written in OMG IDL therefore completely define the operations that a client may request of an object (Mowbray & Ruh 1997). From the interface definition, the IDL compiler generates type information for each method in an interface, and stores it in the interface repository. The IDL compiler also generates the **client stub** and implementation **skeleton** by which a client can invoke a local function. An invocation then occurs on an object on another machine.

The client *stub* is the local object through which the client makes requests. The IDL compiler generated stub provides the correct mapping to the target language, code to locate the skeleton, and marshalling code

necessary to encode parameters. The skeleton is the implementation-side equivalent of the stub. The IDL compiler generated skeleton provides the correct language mapping for invocation to the object, and code for unmarshalling encoded parameters. Clients are not written in OMG IDL, which is purely a descriptive language, but in languages for which mappings from OMG IDL concepts have been defined. The mapping of an OMG IDL concept to a client language construct will depend on the facilities available in the client language. For example, an OMG IDL exception might be mapped to a structure in a language that has no notion of exception, or to an exception in a language that has. Different IDL mappings are defined by the OMG, discussed in the next paragraph.

(i) IDL Language Mappings

Clients written in C, C++, SmallTalk, Cobol, Ada, or Java, can call IDL interfaces through an IDL language mapping (Orfali *et al.* 1996b). This IDL language mapping translates the IDL concepts to the client language, allowing communication between the client and any object reachable by the ORB. The first IDL language mapping, namely C, was provided by the OMG after defining CORBA 1.1. Consequently, any object implementation conforming to OMG, can be called from C. Mappings to C++, SmallTalk, Cobol, and Ada, was introduced in CORBA 2.0, while CORBA 2.2 added the Java IDL mapping. In the new version 3.0 of CORBA, support was added for LISP. Non standard mappings are currently available for Visual Basic, Objective C, and Perl.

The OMG's IDL is considered as one of the key elements of CORBA, since it provides an interface definition of the functionality of all objects implemented on the ORB. In the next section, two ways of object invocation in CORBA are outlined, namely static and dynamic invocation.

3.4.4 Static and Dynamic Invocation in CORBA

Two types of invocation are supported in CORBA, mainly as the result of two separate **Request for Proposals (RFPs)** (Orfali *et al.* 1996b). The first RFP, developed by HyperDesk and Digital, was based on *dynamic invocation*. The second RFP, developed by Sun Microsystems and HP, was based on *static invocation*.

With **static invocation**, the definition of the object's interface is pre-compiled by an IDL compiler. The compiler generates both the stub and skeleton for the client and server platforms. The client, when initiating a request, passes its request through the interface stub to the ORB core. The ORB core transports the request to the server platform, where the request is passed to the skeleton with the help of the Object Adapter (Baker 1997). Finally, the skeleton passes the request to object implementation, where the actual invocation takes places. The results are then passed back to the client, following the same route.

With **dynamic invocation**, an object's interface is not known at compile-time, and must therefore be discovered at run-time. The interface repository allows run-time querying of object's interfaces. A client can thus query the interface repository to get run-time information about a particular interface, and then use that information to send a request through the dynamic invocation interface to the ORB core. The ORB core transports the request to the server platform, where the request is passed to the dynamic skeleton interface with the help of the Object Adapter. The DSI then queries the implementation repository to locate and activate an implementation of the object. Finally, the DSI passes the request to the object implementation. With dynamic invocation, the stub and skeleton is therefore respectively replaced by the DII and DSI.

In the next two sections, CORBAservices and CORBAfacilities, introduced in *section 3.3.2* as part of the Reference Model, are discussed.

3.5 CORBAservices

The initial focus of the OMG standardisation effort was the ORB, providing the basic communication channel through which distributed objects interact. After its completion, the OMG also started standardising the low-level functionality needed by objects, such as persistence, naming, directory, transaction, etc (Mowbray & Ruh 1997). These system-level services, called CORBAservices, are defined in the **Common Object Service Specifications (COSS)**, outlined in the next section.

3.5.1 Common Object Service Specifications (COSS)

The process used by the OMG to create Object Service specifications is based on RFPs (Orfali *et al.* 1996b). Five different groups of RFPs, called COSS, are summarised in *Table 3.2*.

COSS	RFP Issue Date (M/Y)	OMG Adoption Date (M/Y)	Services
COSS1	10/1992	2/1994	Life cycle, Naming, Persistence, and Event Notification
COSS2	7/1993	12/1994	Transactions, Concurrency, Relationships, and Externalisation
COSS3	8/1994	12/1995	Security and Time
COSS4	6/1994	10/1995	Query, Licensing, and Properties
COSS5	8/1995	11/1996	Trader, Version and Collection

Table 3.2: The Common Object Service Specifications

[derived from (Orfali *et al.* 1996b:61) Table 3.1]

Although new CORBA versions have been published since 1997, the COSS services remained constant. In the following paragraphs, each service is discussed.

(i) Life Cycle Service

The life cycle service defines operations for creating, deleting, copying and moving distributed objects.

(ii) Naming Service

The naming service provides the ability to bind a name to an object relative to a *naming context*. A naming context is an object that contains a set of name bindings in which each name is unique.

(iii) Persistent Object Service

The persistent object service provides a set of common interfaces to the mechanisms, such as object-oriented databases or files, used for retaining and managing the persistent state of objects.

(iv) Event Service

The event service allows components to register or de-register dynamically their interest in a specific event.

(v) Transaction Service

The transaction service allows the management of transactions, either supporting the flat or nested transaction models. It also supports the 2PC protocol to be used with reparable objects, that is objects for which the effects of a transaction can be cancelled provided they have not been committed.

(vi) Concurrency Control Service

The concurrency control service enables multiple transactions to co-ordinate their access to shared resources, by means of a lock manager.

(vii) Relationship Service

The relationship service allows relationships and *roles* to be explicitly represented, enforcing referential integrity constraints. A role represents a CORBA object in a relationship.

(viii) Externalisation Service

The externalisation service defines protocols and conventions for externalising and internalising objects. Externalising an object is to record the object's state in a stream of data, in memory or on a disk file, and send it across a network. On the other side it is then internalised into a new object, with the same or a different process.

(ix) Security Service

The security service is incorporated into the ORB as a mechanism for authentication, based on encrypted passwords.

(x) Time Service

The time service allows the management of global time for the complete distributed object system. This incorporates the means for synchronising clocks should the drift exceed a given limit.

(xi) Query Service

The query service allows users and objects to invoke queries on collections of other objects. Queries are based on the SQL-92 standard, the upcoming SQL-3 standard; and the **Object Database Management Group (ODMG)** object query language standard. The ODMG is a non-profit consortium of vendors and interested parties, founded in 1991 with the aim of producing a set of standards for OODBs (CORBA 1995).

(xii) Licensing Service

The licensing service provides a mechanism for producers to control the use of objects, and charge usage per session, per node, per instance creation, and per site.

(xiii) Property Service

The property service enables descriptive attributes to be added dynamically to objects.

(xiv) Object Trader Service

The object trader service allows the identification, description, and promotion of suppliers of services on networks that are compatible with CORBA.

(xv) Version Management Service

Version management allows the support of different versions of components, especially where the interface repository is concerned.

(xvi) Object Collections Service

The purpose of the object collection service is to provide a uniform way to create and manipulate the most common collections generically. Collections are groups of objects, which as a group, support some operations and exhibit specific behaviours that are related to the nature of the collection rather than to the type of object they contain. Examples of collections include sets, and queues.

3.6 CORBAfacilities and CORBAdomains

Unlike CORBAservices, which was the standardisation of low-level functionality needed by objects, CORBAfacilities (discussed in *section 3.3.2*) provides the standardisation of high-level functionality. This was accomplished by the OMG by utilising RFPs to create specifications for functionality usable by almost every application, such as compound document control. A traditional document is a monolithic block of data inside a file, controlled by a single application. In contrast, a *compound document* consists of different types of data like sound, text, and video, each controlled by its own software. Orfali *et al.* (1996a:428) defines a **compound document** as *'a container composed of different kinds of software components and data.'* As the name implies, a container defines the outermost document, the one that contains other objects. Currently, the CORBAfacilities specification consisting of four basic services, including *User Interface Management*, *Information Management*, *System Management*, and *Task Management*. These services are listed in *Table 3.3*.

Service	Services
User Interface Management	Provide facilities for making an information system accessible to its users.
Information Management	Enable the modelling, defining, storing, and retrieving of information.
System Management	Provides facilities for the management of complex multi-vendor information systems.
Task Management	Enable the automation of work processes.

Table 3.3: The Four Basic Services of CORBAfacilities

Like CORBAfacilities, CORBAdomains are also an OMG specification for high-level functionality needed by objects. The CORBAdomains are however standards for interoperability in particular industrial areas. The most important of these include finances, telecommunications, manufacturing, and healthcare.

3.6.1 Reuse of specifications

Much of the interest in OO is motivated by the promise of reusability. A *framework* is a major element in enabling reusability in applications. Harmon and Morrissey (1996) define a **framework** as a *'group of objects designed to support specialised programming functions like compound documents, databases access, etc.'* This is precisely what the CORBAservices, CORBAfacilities, and CORBAdomains specifications form (discussed in *section 3.3.2*), when implemented in a CORBA product (Bouzeghoub *et al.* 1997). Consisting of groups of objects with interfaces defined in OMG's IDL, they provide reusability through inheritance as illustrated in *Figure 3.3*.

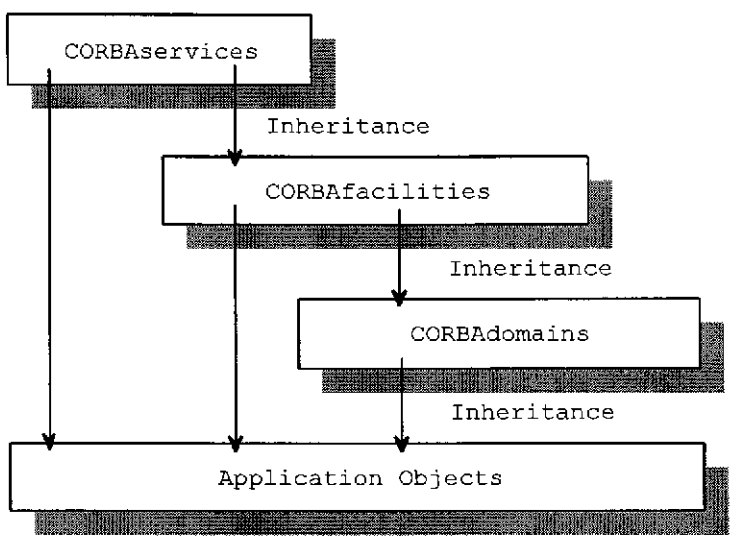


Figure 3.3: Reusability through inheritance in CORBA

[derived from (Bouzeghoub *et al.* 1997:95) Figure 4.3]

CORBAservices and its fundamental capabilities are reused and inherited throughout the CORBA architecture, in particular by **CORBAfacilities**, which specialisation is globally applicable across vertical domains. CORBAservices and CORBAfacilities are again reused by **CORBAdomains**, which provide the next level of specialisation. The final level is **application objects**, which reuse specifications from all areas, including CORBAservices, CORBAfacilities, and CORBAdomains.

The separation between application objects from CORBAfacilities and CORBAservices are not static, but reflect the evolution of object technology (Orfali *et al.* 1996b). The current placement therefore reflects the present OMG standardisation effort. As experience in CORBAdomains matures, areas of potentially new CORBAfacilities are discovered and defined. CORBAfacilities, which have become fundamental to all or

most CORBA implementations, can again be incorporated by the OMG into CORBAservices. The CORBA specification, especially CORBAservices, CORBAfacilities, and CORBAdomains, can therefore change regularly depending on user needs.

3.7 Object Request Broker Interoperability and CORBA 3.0

The CORBA 1.0 specification focussed on the creation of distributed object applications, leaving the implementation of the object request broker to the vendors. This resulted in some level of component portability, but without real ORB interoperability. Different CORBA 1.0 products could therefore not communicate over heterogeneous environments (Bouzeghoub *et al.* 1997). The CORBA 2.0 specification however added specific protocols and operations to allow communication between different ORBs over heterogeneous environments, specifically the **General Inter-Orb Protocol (GIOP)**, the **Internet Inter-Orb Protocol (IIOP)**, and **Environment Specific Inter-Orb Protocols (ESIOPs)**. In the next three sections, the functions of each of these protocols are described in more detail. The next version of the CORBA standard, named CORBA 3.0, is also introduced.

3.7.1 General Inter-ORB Protocol (GIOP)

The general inter-ORB protocol is a protocol enabling requests and responses to be exchanged over an arbitrary transport protocol (CORBA 1995). It is simple and defines at the same time a low-level data representation, as well as a set of message formats for communication. The low-level data representation imposes a common representation for the data types of the CORBA object model. The various message types and formats that this protocol supports are presented in *Table 3.4*.

Message	Origin	Role
Request	Client	Sends request.
CancelRequest	Client	Cancels request issued.
LocateRequest	Client	Finds location of an object.
Reply	Server	Return response to a request.
LocateReply	Server	Response to location request.
MessageError	Client, Server	Response to an invalid message.

Table 3.4: The Seven Message Types of the General Inter-ORB Protocol

[derived from (Bouzeghoub *et al.* 1997:296) Figure 9.1]

Request sends a request, while *CancelRequest* cancels the request (Bouzeghoub *et al.* 1997). A corresponding message from the server, namely *Reply*, sends the response to the client. Two messages are for locating objects, which may have migrated or been destroyed. The first is *LocateRequest*, issued by

the client, and the second, *LocateReply*, returned by the server. The location request can also be used to find whether an object reference is valid, whether the server is able to deal with a request for the object and, if not, to what address should the request be sent. Finally, a message signalling an error, namely *MessageError*, can be sent by either the client or the server.

3.7.2 Internet Inter-ORB Protocol (IIOP)

The Internet inter-ORB protocol defines how messages conforming to GIOP can be sent over the Internet (Bouzeghoub *et al.* 1997).

3.7.3 Environment Specific Inter-ORB Protocols (ESIOP)

Environment specific inter-ORB protocols are intended for specific environments, hence the general name and the acronym ESIOP (Bouzeghoub *et al.* 1997). The first of these protocols is the **Distributed Computing Environment Common InterOperability Protocol (DCE CIOP)**, specifically for the OSF's DCE, discussed in *Chapter 2*. DCE CIOP allows ORBs to be implemented over multi-vendor platforms, mainly as a result of the success of the DCE standard, and the availability of RPC based middleware. *Figure 3.4* illustrates the relationship between GIOP, IIOP and DCE CIOP.

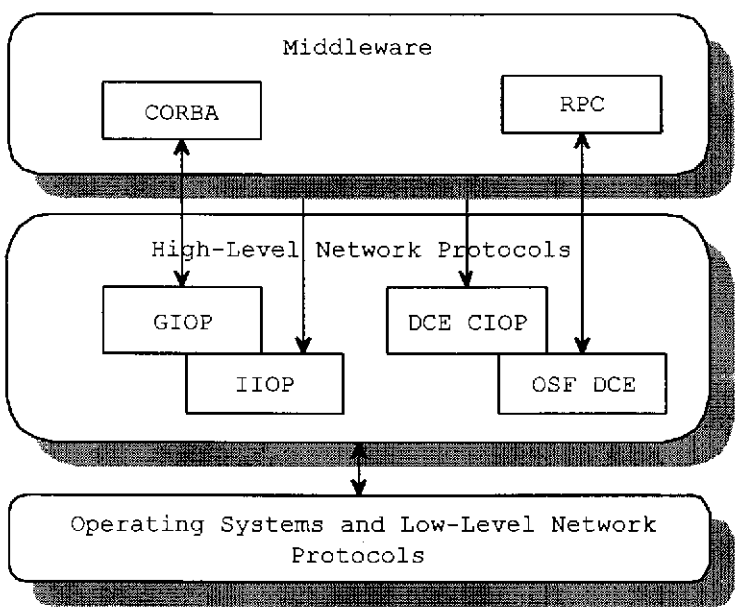


Figure 3.4: The relationship between GIOP, IIOP and DCE ESIOP

[derived from (Bouzeghoub *et al.* 1997:107) Figure 4.9]

By implementing GIOP, IIOP, or ESIOPs, most ORB vendors can ensure their products interoperability with other vendors. The OMG has also lately introduced the Portable Object Adapter specification. This specification let clients written to access an ORB from one vendor, easily access other vendor's products as well.

3.7.4 CORBA 3.0

CORBA 3.0 is the successor to CORBA 2.3, announced in 1999, adding a number of new technologies to the CORBA specification (Siegel 1999). The most important of these technologies include *Internet integration*, *asynchronous messaging*, and the *CORBAcomponent architecture*, outlined below.

- The **Internet integration** technology consists of the **Interoperable Name Service (INS)**, providing a URL to object reference mechanism.

- The **asynchronous and messaging invocation** specification (quality of service control), defines a number of asynchronous and time-independent invocation modes for CORBA, and allows both static and dynamic invocations to use every mode.

- The **CORBAcomponents** specification is one of the most exciting new developments in CORBA 3.0. The main reason for this is that it provides a container environment that packages transactionality, security, and persistence, a scripting language, and provides interface and event resolution for distributed objects, also called components. This is basically the same functionality provided by the **Microsoft Transaction Server (MTS)**, outlined in the next chapter.

At this point in time, the CORBA 3.0 specification has not yet been fully implemented, making it difficult to assess the full impact on current CORBA products. Backward compatibility is however ensured by OMG, making any large architecture changes between CORBA 2.x and CORBA 3.0 negligible.

3.8 Summary

In this chapter, CORBA is highlighted as one of the most popular and comprehensive object-oriented middleware standards. Developed by the OMG, it is based on the OMG OMA, consisting of the Core Object Model and Reference Model. Another important OMG specification, namely the OMG IDL, is also highlighted. Used to define all interfaces to the ORB, CORBAservices, CORBAfacilities, and CORBAdomains, it plays a crucial role in CORBA. The main elements of CORBA architecture, including the *client*, *object implementation*, *ORB Core*, *stub*, *dynamic invocation interface*, *skeleton*, *dynamic skeleton interface*, *Object Adapter*, *ORB interface*, *interface repository*, and *implementation repository*, are also defined, specifying the role of each element in providing distributed object communication. Although CORBA is currently considered to be the most dominant standard for distributed object computing, it is losing ground to Microsoft Corporation's DCOM, mainly as a result of Microsoft's control of the operating system market. In the next chapter, the DCOM object-oriented middleware standard is outlined in order to get a better understanding of its functionality.

Chapter 4 : Distributed Component Object Model (DCOM)

4.1 Introduction

The previous chapter expounded the CORBA object-oriented middleware standard. In this chapter, its main competitor, Microsoft Corporation's **Component Object Model (COM)/Distributed Component Object Model (DCOM)** is discussed. COM is an integration infrastructure, or middleware, used to implement objects that interact within a single address space (called *in-process*) or between processes on a single hardware platform (called *local* or *out-of-process*). This implies that COM serves as the middleware in a non-distributed environment. Currently, COM forms the foundation of **Object Linking and Embedding (OLE)**, and an increasing number of other services provided by Microsoft's operating systems (Chappell 1996). DCOM is essentially an extension of COM, referred to as *'COM on a long wire'*, providing COM objects the ability to communicate between different hardware platforms over a network. DCOM is therefore the middleware used by Microsoft in a distributed environment.

In the following sections, the main elements of the COM/DCOM standard are discussed. These include OLE, COM, the COM/DCOM architecture, Microsoft's Interface Definition Language, COM services, COM/DCOM interoperability, and COM+. The developer and main driving force behind DCOM, namely Microsoft Corporation, is introduced in the following section.

4.2 Microsoft Corporation

If CORBA is considered to be the leading object-oriented middleware standard, then Microsoft Corporation's DCOM is the *de facto* other standard. What makes DCOM so important? The answer is Microsoft Corporation, the global giant in software development. All new software developed by Microsoft Corporation today is based on DCOM, thereby having a mondial impact on personal computers and software systems. This strategy, announced in May 1995, was seen as the first major move to distributed object computing by a software developer, essentially building its operating systems and application software as distributed objects (Rogerson 1997).

In July 1996, Microsoft also announced the transition of the OLE, COM/DCOM, and ActiveX specifications to an industry-standards body, namely The Open Group. This resulted in The Open Group forming **The Active Group** in October 1996, with the main aim to manage the evolution of these technologies. DCOM must therefore be considered as the major competitor of CORBA, supported by Microsoft, and implemented

in most of its products. In the next section, OLE is explored, being the first product developed by Microsoft to utilise COM.

4.3 Object Linking and Embedding (OLE)

The development of COM/DCOM has its roots in OLE. The first version of OLE, namely OLE 1, was introduced in 1990 by Microsoft Corporation as a technology to support the linking and embedding of objects created in one application, called a *server*, into a document in another application, called a *container* (Orfali *et al.* 1996a). OLE 1 is therefore classified as a **compound document framework**, defined by Harmon and Morrissey (1996) as *'a desktop centric framework, acting as a container for different types of components, providing basic interaction functionalities.'*

In 1993, Microsoft Corporation introduced a subsequent version, called OLE 2, solving many of OLE 1's shortcomings with a new infrastructure technology (Brockschmidt 1993). This new technology, namely COM, grew out of the desire to provide a common paradigm for interaction among all types of software, including libraries, applications, and systems software. Accordingly, while OLE 2 was the first technology to use COM, it wasn't really tied to compound documents in any significant way. Very soon, COM was used in technologies that had nothing whatsoever to do with compound documents. Unfortunately, Microsoft Corporation's marketing department decided to use OLE, without a version number, as the brand name for COM. This resulted in confusion between OLE and COM, the one being a compound document framework, the other middleware.

In 1996, Microsoft Corporation introduced DCOM, extending COM from a non-distributed middleware standard to a distributed middleware standard. This was accomplished by implementing the Distributed Computing Environment RPC protocol in COM, allowing communication between different hardware platforms. Previously COM only supported the **Lightweight Remote Procedure Call (LRPC)** protocol, used for inter-process communication on the same hardware platform. In 1996, Microsoft Corporation also announced its Internet strategy, changing brand names once again. A new brand name for COM-based technologies was chosen, namely ActiveX, and OLE was again deemed to refer only to compound documents. COM/DCOM therefore forms the basis of ActiveX, the technology implemented in all Microsoft's Internet products. In the next section, a short overview of the OLE architecture is given.

4.3.1 OLE Architecture

Microsoft Corporation's OLE architecture consists of two higher-level services, namely *OLE Compound Documents* and *OLE Automation*, build upon COM, illustrated in *Figure 4.1* (Orfali *et al.* 1996a).

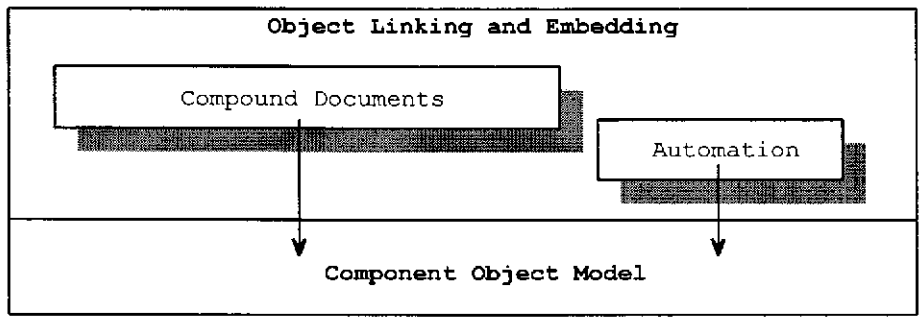


Figure 4.1: OLE Architecture

[derived from (Orfali *et al.* 1996a:453) Figure 26.1]

These services therefore exists separately from COM, outlined in the following two paragraphs.

(i) OLE Compound Document Service

Built on top of COM services, OLE Compound Documents support the creation and management of compound documents. Two primary components of OLE Compound Documents can be identified, namely the *container* and *Compound Document objects* (Orfali *et al.* 1996a). The **container** is the component that controls the document and manages the relationships between the pieces of information in the document. **Compound document objects** are the pieces of information created by specific applications. Containers can include Compound Document objects either by *embedding* or *linking*. With *embedding*, the entire object is embedded within the container. With *linking*, a representation of the compound document object is cached in the container together with a reference to the location of the object.

(ii) OLE Automation Service

Another key part to integrating objects is the ability to drive them programmatically, that is, to control them without requiring any end user's intervention. In more technological terms, it means having various objects, each exposing its end-user level functionality via interfaces. These interfaces can then be manipulated by using a scripting tool, invoking specific functions of the object.

Although OLE played an important role in the development and initial usage of COM, it quickly became more than just an enabling technology for OLE. In the next section, the COM is enunciated in more detail.

4.4 Component Object Model (COM)

At the core, the Component Object Model is a specification, documented by Microsoft Corporation in the COM Specification. This specification, or Object Model, defines how objects and their clients interact through the binary standard of interfaces (Chappell 1996). As a specification it also defines a number of other standards for interoperability, commonly known as COM services.

In addition to being a specification, COM is also an implementation. This implementation, generally referred to as the COM Library, is utilised by COM to enable communication between objects. It is important to note that while the CORBA standard only consists of a **specification**, the COM standard consists of a **specification** and **implementation**. In the following sections, the *COM Object Model* and *COM/DCOM Architecture* are further explored. Attention is also given to *Microsoft's Interface Definition Language*, and static and dynamic invocation in COM/DCOM.

4.4.1 COM Object Model

COM is designed to allow clients to transparently communicate with objects regardless of where those objects are executing, be it the same process, the same machine, or a different machine. What this means is that there is a single object model for all types of objects, be they clients or servers of these objects. This *binary* object model allows COM to intercept an interface call to an object, and instead make a remote procedure call to the instance of the object that is running in another process or on another machine (Rogerson 1997). While there is a great deal more overhead in making a remote procedure call, no special code is necessary in the client to differentiate an in-process object from out-of-process objects. All objects are available to clients in a uniform, transparent fashion.

In the next few paragraphs the more important elements of the COM Object Model are highlighted, including *COM objects* and *interfaces*, *COM classes*, the *IUnknown* interface, and *exceptions*. Also discussed is the concepts *multiple interfaces*, *inheritance*, *encapsulation*, and *polymorphism*.

(i) COM Objects and Interfaces

A COM object is an instance of a user or system defined COM class, and defines a group of interfaces, each of which includes a number of methods, illustrated in *Figure 4.2*.

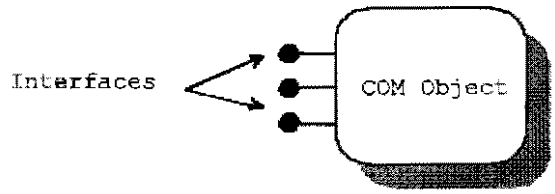


Figure 4.2: Component Object Model (COM) Object

[derived from (Chappell 1996:7) Figure 1.3]

Three important characteristics of COM interfaces that can be identified are the following (COM 1995):

- A COM interface is a strongly typed **contract** between objects to provide a small but useful set of semantically related functions. Note that all COM and OLE services are simply implemented as a grouping of COM interfaces.

- The name of a COM interface is always prefixed with an 'I' by convention, as in *IUnknown*. Every interface however has its own unique identifier, called an **Interface IDentifier (IID)**, thereby eliminating any chance of clashes that could occur with normal names. IIDs are really **Globally Unique IDentifiers (GUIDs)**, 128-bit integers that are virtually guaranteed to be unique in the world across space and time. These GUIDs are the same as UUIDs used by OSF's DCE, discussed in *Chapter 2*. Classes in COM are also uniquely identified by GUIDs, called **ClaSs IDentifiers (CLSIDs)**.

- Interfaces are **immutable**, and can therefore never be changed to support new functionality. To create a new version of an interface, either by adding or removing functions or changing semantics, an entirely new interface must be created with a new IID. Therefore, a new interface never conflicts with an old interface, even if nothing has changed except the semantics.

(iii) COM Classes

A COM class is an implementation of a set of COM interfaces (COM 1995). Each COM class is identified by a unique 128-bit class identifier, which associates an object class with a particular implementation. Class identifiers are generally obtained through the *CoCreateGUID* function in COM, or through a COM-enabled tool that internally calls this function. The use of unique CLSIDs avoids name collisions between COM classes, since CLSIDs are in no way connected to the names used in the underlying implementation. For example, two different vendors can write COM classes, called *ArrayClass*, but each will have a unique CLSID and therefore avoid any possibility of a clash.

(iv) The IUnknown Interface

All COM objects must implement one important interface. This interface, namely **IUnknown**, consists of three methods, called **Release**, **AddRef**, and **QueryInterface**, used to create, delete, and find interfaces in COM (Chappell 1996). The IUnknown interface therefore forms the key in managing COM objects. Just like an application must free allocated memory which is no longer in use, a client of a COM object is responsible for freeing the object when that object is no longer required. In COM, the client can only do this by giving the COM object an instruction to free itself. However, the difficulty arises when the COM object knows that it is safe to free itself. This is particularly the case for COM objects that may be in use by multiple clients at the same time. The COM object must then wait until *all* clients have finished using it, before it can free itself.

COM specifies a **reference counting** mechanism to provide this control. Each COM object maintains a 32-bit reference count that tracks how many clients are connected to it, that is, how many pointers to any of its interfaces exist. The two methods **AddRef** and **Release** manage reference counting in COM. When a COM object is *created*, that is, when the first interface pointer to the object is created, the reference count is incremented by one. With each new reference to the object, the *AddRef* method must be called, incrementing the reference count by one. With each reference *deletion*, the *Release* method must be called, decrementing the reference count by one. While the object's reference count is nonzero, it must remain in memory, when the reference count becomes zero, the object can safely be unloaded since no other objects hold references to it. The **QueryInterface** method is the mechanism whereby a client, having obtained one interface pointer to a particular COM object, can request additional pointers to other interfaces of that same COM object. One input parameter to the *QueryInterface* method is the interface identifier of the interface being requested. If the COM object supports the interface, it returns the appropriate interface pointer to the client, if not, the COM object returns an error.

(v) Exceptions

COM/DCOM do not support exceptions (COM 1995). Instead, COM uses a specific *return value* to indicate status and error information of the interface call. This return value, called **HRESULT**, is a simple 32 bit value divided into fields indicating success or error, and specific status codes.

(vi) Multiple Interfaces

In COM, an object can support multiple interfaces, that is, provide pointers to more than one virtual function table (COM 1995). Multiple interfaces are a fundamental innovation of COM to avoid versioning problems and any strong association between an interface and an object class.

(vii) Delegation and Aggregation

Inheritance, one of the object-oriented concepts discussed in *Chapter 2*, is not supported in COM. COM however defines two concepts, namely *delegation* and *aggregation*, serving as Microsoft's form of inheritance (Chappell 1996). For convenience, the object being reused is called the *inner object*, and the object making use of that inner object is the *outer object*. With **delegation**, depicted in *Figure 4.3*, the outer object behaves like a client of the inner object. When the outer object wishes to use the services of the inner object, the outer object simply *delegates* the implementation to the inner object. For example, when the outer object receives a request for interface B, the outer object *IUnknown* interface delegates the request to the inner object.

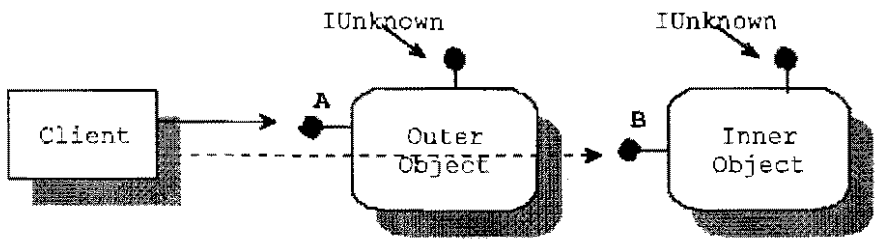


Figure 4.3: Delegation in COM

[derived from (Chappell 1996:65) Figure 2.7]

With **aggregation**, the outer object exposes the interfaces of the inner object as if they were part of its own *IUnknown* interface, as depicted in *Figure 4.4*. The outer object therefore aggregates the inner object. For example, the interface B is part of the outer object's IUnknown interface.

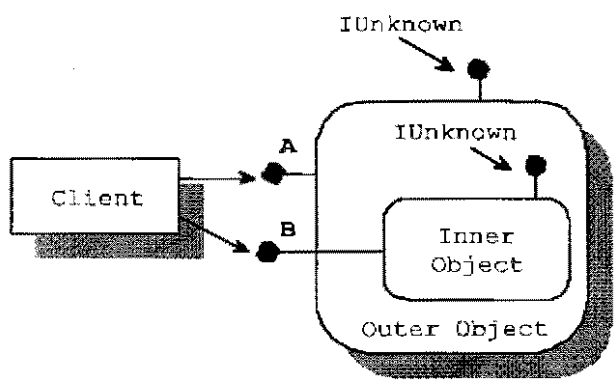


Figure 4.4: Aggregation in COM

[derived from (Chappell 1996:66) Figure 2.8]

Since inheritance is not supported in COM, no inheritance hierarchy with superclasses and subclasses can be implemented. The COM environment is therefore inherently flat, implementing reusability through a web of pointers that link or aggregate different interfaces.

(viii) Encapsulation

COM supports encapsulation by prohibiting direct access to an object's implementation and data (Rogerson 1997). All access to COM objects are by interfaces, thereby hiding the implementation and data of objects.

(ix) Polymorphism

All COM interfaces support polymorphism, thereby implying that when calling a function using an interface pointer, the implementation invoked is not specified. In the next section, the main elements of the COM/DCOM architecture are described in more detail.

4.4.2 The COM/DCOM Architecture

In *Figure 4.5*, the main elements of the COM/DCOM architecture are presented, including the *client*, *object implementation*, *proxy*, *stub*, *COM library*, *service control manager*, and *registry*.

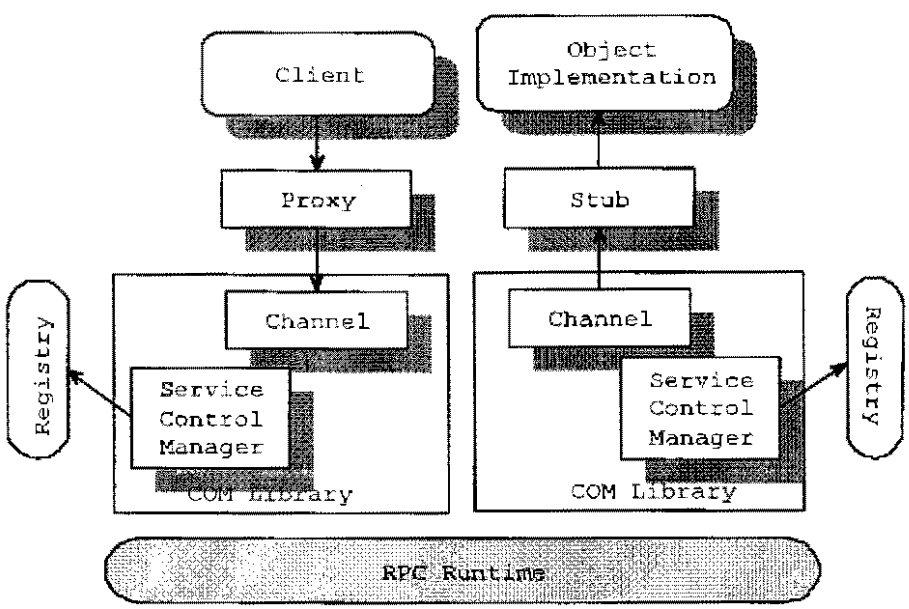


Figure 4.5: The main COM/DCOM elements

[derived from (Chappell 1996:253) Figure 10.6]

The interaction of the elements of the COM/DCOM architecture is explained later in the chapter, specifically when static and dynamic invocations in COM/DCOM are discussed. The following paragraphs provide a short outline of each element.

(i) Client

A COM client is a COM object that makes use of another COM object through that object's interfaces. The COM client can be on the same or a distributed platform.

(ii) Object Implementation

The object implementation is the code that actually implements services, requested by COM objects.

(iii) Proxy

The *proxy* is a piece of interface-specific code that resides in the client's process space and prepares the interface parameters for transmittal. It packages, or marshals them in such a way that they can be recreated and understood in the receiving process.

(iv) Stub

The *stub* is a piece of interface-specific code that resides in the server's process space and reverses the work of the *proxy*. The stub unpackages, or unmarshalls the sent parameters, and forwards them on to the server. It also packages reply information to send back to the client.

(v) COM Library

Every hardware platform that provides support for COM must implement the COM Library (Chappell 1996). Consisting of a standard set of API functions, it also provides support for communicating between COM objects. Three of the most important elements of the COM Library are the **Service Control Manager (SCM)**, *Type Library*, and *channel*:

- The *service control manager* ensures that when a client request is made, the appropriate server is connected, ready to receive the request. The SCM keeps a cache list of CLSIDs, called the **class object table**, for the object implementations on the local machine, based on the *registry*. This is the foundation for COM's locator service.

- The *Type Library*, also called an interface header file, contains the type information of all supported interfaces and their methods in COM.

- The *channel*, with the help of the RPC runtime library, is responsible for the transportation of data across different networks. On the client side, the client's method call goes through the *proxy* and then onto the channel. The channel sends the buffer containing the marshalled parameters to the RPC runtime library, which transmits it across the network. On the server side, the RPC runtime library sends the client's method call to the appropriate channel, which passes it to the *stub*. The *stub* then unmarshalls the call, passing the required information onto the object implementation.

The COM Library is therefore the underlying plumbing that makes everything work transparently through RPC.

(vi) Registry

The registry provides the same service for COM as the implementation repository for CORBA, by associating the CLSID and the path name of the server object. In other words, the registry serves as a persistent store of *CLSID-to-server* mappings that it uses to implement a locator service in COM.

4.4.3 Microsoft's Interface Definition Language (MIDL)

The **Microsoft Interface Definition Language (MIDL)**, the IDL used and supplied by Microsoft Corporation, is based on simple extensions to the OSF's DCE interface definition language (COM 1995). When defining custom interfaces for a COM object, a developer can create an interface definition using MIDL, describing the data types and methods of these interfaces. From this interface definition, the MIDL compiler generates the source code necessary to build the *proxy* and *stub* for the COM object, registered in the system registry, as well as the Type Library. MIDL is however a tool of convenience and not central to COM's interoperability. It essentially saves the developer from manually creating proxies, stubs, and the Type Library entries by hand. In the next section, two ways of object invocation in COM/DCOM are discussed, namely static and dynamic invocation.

4.4.4 Static and Dynamic Invocation in COM/DCOM

Like CORBA, COM objects can be invoked either statically or dynamically (Chappell 1996).

With **static invocation**, the client knows the class identifier of the object to be created, and the interface identifiers of interfaces that the object supports. When creating a single object, the client first calls the COM Library function *CoCreateInstance*, which then delegates the request to the service control manager (see Figure 4.5). The SCM uses the class identifier to find the entry for this object's class in the registry. This entry specifies the location of the server capable of instantiating the specified object class. Once the server is found, the SCM starts it.

With the CLSID and the IID, the *CoCreateInstance* function also allows the client to specify the kind of server COM should start. Three types of COM servers can be identified, namely in-process, local, and remote, as illustrated in *Figure 4.6*. **In-process** servers can be loaded directly into the client's process space, serving in-process objects. Under Microsoft Windows and Microsoft Windows NT, these are implemented as **Dynamic Link Libraries (DLLs)**. **Local servers** are loaded in separate processes on the same machine as the client process, and serves local objects. These servers are normally implemented as **EXEcutable's (EXEs)**, executing in their own processes, as opposed to DLLs, which must be loaded into existing processes. **Remote servers** are loaded on remote machines, executing in totally different processes. Remote servers may be implemented as either DLLs or EXEs.

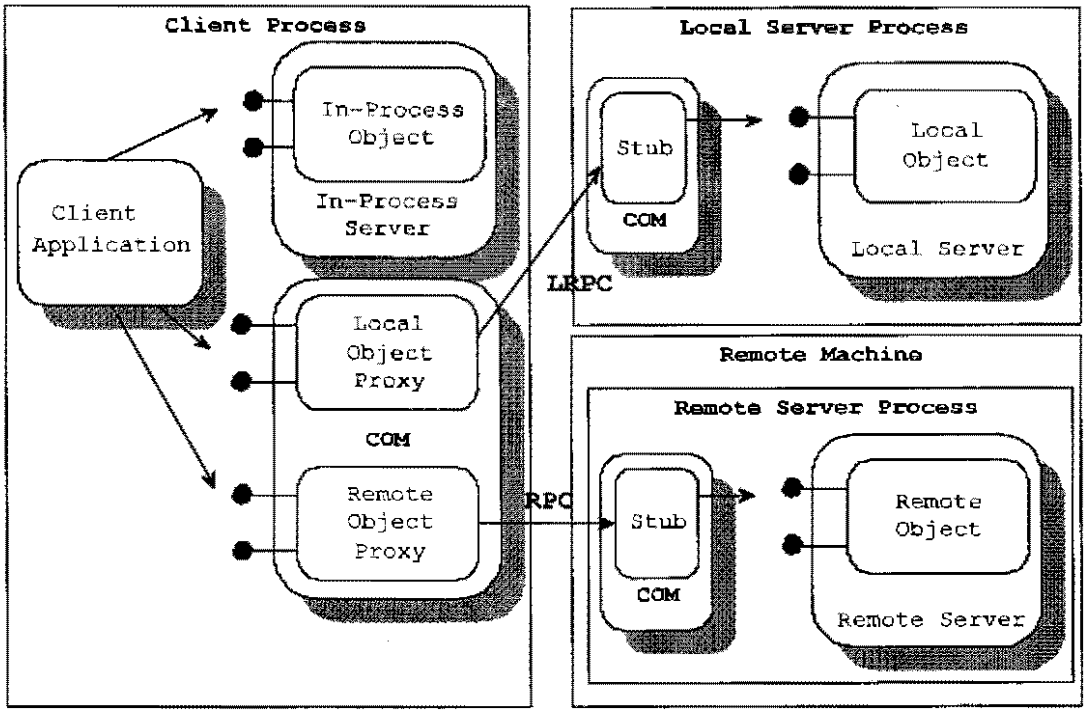


Figure 4.6: In-Process, Local, and Remote COM server

[derived from (Orfali *et al.* 1996a:455) Figure 26.2]

If the client needs only a single instance of a particular object, the simplest solution is to create that instance with *CoCreateInstance*. However, it is possible that a client might need multiple instances of objects of the same class. To create them efficiently, the client can access a *class factory*, defined as *'an object that can create other objects (Chappell 1996).'* Each class factory knows how to create objects of one specific class. Class factories are COM objects, accessed via interfaces, and support the **IUnknown** interface with its **AddRef**, **Release**, and **QueryInterface** methods. In truth, the *CoCreateInstance* also use class factories, although it is hidden from the client.

Each class factory supports the **IClassFactory** interface, which implements two methods, namely **CreateInstance** and **LockServer**. The **CreateInstance** method creates a new instance of the object class that the factory instantiate. The client does not specify a CLSID, since the class of the object being created

is implicit in the factory object itself. The client does however specify an IID, indicating the interface to which it needs a pointer. The **LockServer** method allows a client keep a server in memory, ignoring COM's reference count mechanism.

To access (initiate) a class factory, a client must invoke the COM Library function *CoGetClassObject*, specifying the CLSID of the class of objects the factory will create, and an IID for the **IClassFactory** interface. After initiating the class factory with the *CoGetClassObject* function, the client can call the method **IClassFactory::CreateInstance** of the class factory to create an object. At this point the client has interface pointers for *two separate objects*, the class factory and an object of that class, each having their own reference counts. It is an important distinction that is illustrated in *Figure 4.7*.

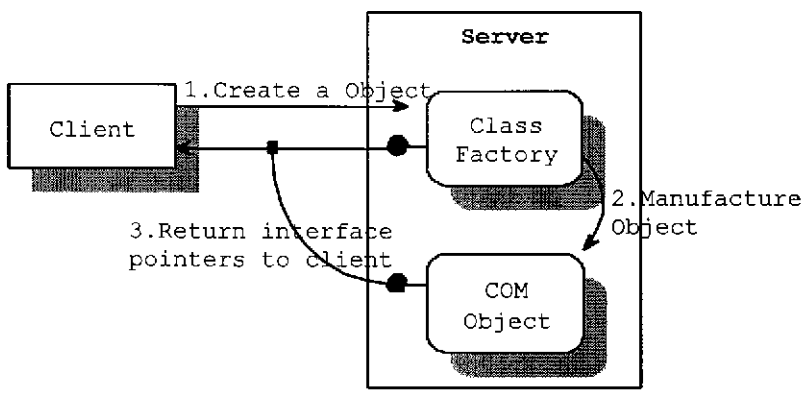


Figure 4.7: A COM Client creates objects using a class factory

[derived from (Chappell 1996:62) Figure 2.6]

The *class factory* must therefore be released, by calling the **IClassFactory::Release** method, just like the COM objects it creates.

With **dynamic invocation**, the client knows the CLSID of the object to be created, but not the IIDs of interfaces that object supports, the methods of these interfaces, or parameters required. To access this information, the client must access the Type Library, i.e. COM's version of CORBA's *interface repository*. First the client accesses the registry to find the object's CLSID Type Library. Once found, it can use the **ITypeLib** interface, which accesses the type library as a whole, or **ITypeInfo** interface, that accesses the individual objects in the library. Since no proxy or stub objects are created specific to these interfaces, COM utilises the **IDispatch** interface, serving the same purpose as CORBA's *dynamic invocation* and *dynamic skeleton interface*. In the next section, the main COM services are enunciated.

4.5 COM Services

COM services are a collection of services that provide general supportive functionality for COM objects (COM 1995). These include two services previously classified as part of OLE, namely *persistent storage*

and *uniform data transfer*, and other services like *life cycle*, *monikers*, *transaction*, *query*, *event*, *licensing*, *security*, and *messaging*, discussed in the following sections.

4.5.1 Life Cycle Service

The life cycle service is implemented in COM by the **AddRef** and **Release** methods of the **IUnknown** interface (COM 1995). When a COM object is *created*, that is, when the first interface pointer to the object is created, the reference count is incremented by one. With each new reference to the object, the *AddRef* method must be called, incrementing the reference count by one. With each reference *deletion*, the *Release* method must be called, decrementing the reference count by one. While the object's reference count is nonzero, it must remain in memory, when the reference count becomes zero, the object can safely be unloaded since no other objects hold references to it.

In COM, objects are periodically pinged, to ensure that they are still active. In the distributed world of DCOM, this would be very wasteful, since it generates vast amounts of network traffic. To optimise pinging, DCOM uses keep-alive messages on a per-machine basis. That is, independent of the number of COM objects active on a machine, only a single ping message is used between machines.

4.5.2 Persistent Storage Service

COM defines a number of storage-related interfaces, collectively called persistent storage (Rogerson 1997). The most important element of persistent storage is the creation of a file system within a file. Another element is the ability of a COM object to save its state by using the persistent storage service. These elements are described in more detail in the following paragraphs.

(i) Creation of a File System Within A File

A major feature of COM is interoperability, the underlying principle for integration between applications. This integration requires applications to write information to the same file on the underlying file system. This is exactly the same problem that the computer industry faced years ago when multiple applications began to share the same disk drive. The solution then was to create a file system to provide a level of indirection between an application file and the underlying disk sectors. Thus, the solution for the integration problem today is another level of indirection, a **file system *within* a file**. Instead of requiring that a large contiguous sequence of bytes on a disk be manipulated through a single file handle, COM defines how to treat a single file system entity as a structured collection of two types of objects (Chappell 1996).

These objects are **storages** and **streams**, illustrated in *Figure 4.8*, acting like **directories** and **files** respectively.

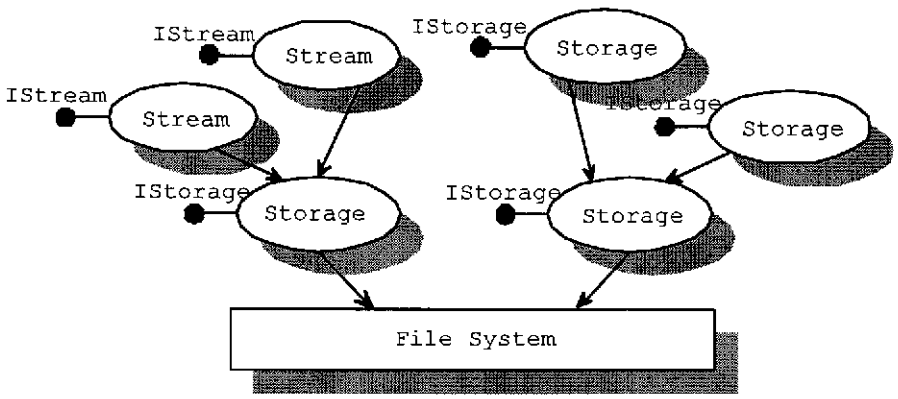


Figure 4.8: Storages and Streams

[derived from (Chappell 1996:26) Figure 1.8]

Like other objects in COM, storages and streams are accessed by interfaces, namely **IStream** for stream objects and **IStorage** for storage objects. The **IStream** interface allows a stream object to read, write, seek, and perform a few other operations on its underlying data. The **IStorage** interface describes the capabilities of a storage object such as listing contents, move, copy, rename, create, and delete.

(ii) Persistent Objects

Because COM allows an object to read and write itself to storage, there must be a way through which the client tells objects to do so. This is accomplished through additional interfaces that form a storage contract between the client and objects. When a client wants to tell an object to deal with storage, it queries the object for one of the persistence-related interfaces, as suits the context (COM 1995). These interfaces, namely **IPersistStorage**, **IPersistStream**, and **IPersistFile**, are described in *Table 4.1*.

Interface	Description
IPersistStorage	An object can read and write its persistent state to a storage object. The client provides the object with an **IStorage** pointer through this interface.
IPersistStream	An Object can read and write its persistent state to a stream object. The client provides the object with an **IStream** pointer through this interface.

Interface	Description
IPersistFile	An Object can read and write its persistent state to a file on the underlying system directly. This interface does not involve **IStorage** or **IStream** unless the underlying file is itself accessed through these interfaces, but the **IPersistFile** itself has no semantics relating to such structures. The client simply provides the object with a filename and orders to save or load.

Table 4.1: The Three Persistence-Related Interfaces

4.5.3 Monikers (Naming Service)

A moniker is simply a persistent object that implements the **IMoniker** interface, used to assign a persistent name to an individual object instance (Chappell 1996). Each moniker object is an instance of a moniker class, which has its own semantics as to what type of object or operation it can reference.

4.5.4 Uniform Data Transfer Service

Just as COM provides interfaces for dealing with storage and object naming, it also provides interfaces for exchanging data between applications (COM 1995). Built on top of both the COM object request broker and the persistent storage service is the uniform data transfer service, which provides the functionality to represent all data transfers through a single implementation of a *data object*. Data objects implement an interface called **IDataObject,** which encompasses the standard operations of get/set data as well as functions through which a client of a data object can establish a notification loop to detect data changes in the object.

4.5.5 Transaction Service

COM/DCOM does not support the transaction service, which allows the management of transactions. Microsoft Corporation however developed two products, namely the **Microsoft Transaction Server (MTS)** and **Distributed Transaction Co-ordinator (DTC)**, used for implementing transactions in COM (Orfali et al. 1996b). MTS is an object runtime environment, providing automatic transaction management, concurrency control (thread synchronisation), instance lifecycle management, database session management, and simplified security. DTC is very similar to CORBA's transaction service, implementing the 2PC protocol for transactions.

4.5.6 Query Service

OLE DataBase (OLE DB), the successor of ODBC, is generally considered as COM's query service, although it is implemented as a separate product by Microsoft Corporation (Rogerson 1997).

4.5.7 Event Service

COM/DCOM has a standard event service, which is handled by *connectable objects*. A COM object can support IDL-defined outgoing interfaces as well as incoming ones. These interfaces provide a standard way for defining events and their parameters. Objects that support outgoing interfaces are called connectable objects or sources. A connectable object can have as many outgoing interfaces as it likes. Each interface is composed of a set of outgoing functions, each function representing a single event or notification.

4.5.8 Licensing Service

COM considers a machine to be fully licensed when a *license file* is installed on that machine (COM 1995). Otherwise, the client must supply a special *license key* when any object instance is created. A *license file* is a global permission to use the object on a machine, while the *license key* is a specific permission to use the object on a machine.

4.5.9 Security Service

COM provides security along several crucial dimensions (COM 1995). Firstly, COM uses standard **operating system permissions** to determine whether a client, running in a particular user's security context, has the right to start the code associated with a particular object class. Secondly, with respect to persistent objects, COM uses operating system or application permissions to determine if a particular client can load the object at all, and if so whether they have read-only or read-write access. Finally, because its security architecture is based on the design of the **DCE RPC security architecture**, COM provides cross-process and cross-network object servers with standard security information about the client or clients that are using it. This enables a server to use security in a more sophisticated fashion than that of simple OS permissions.

4.5.10 Message Service

Because COM relies on synchronous RPCs, asynchronous communication is normally not available. COM can however support asynchronous communication with the help of another Microsoft Corporation product, namely **MicroSoft Message Queue (MSMQ)**, which allows an application to send a message to a queue. The message can then be read from the same queue by another application (Orfali et al. 1996b). MSMQ-style communication is very useful when the sender need not wait for a response from the receiver, or if the sender and receiver might not be running at the same time. In the following section, COM/DCOM interoperability and the new COM+ are discussed.

4.6 COM/DCOM Interoperability and COM+

The Object RPC protocol used by COM/DCOM consists of a set of extensions, layered on the distributed computing environment RPC specification (Rogerson 1997). It therefore strongly leverages the Open Software Foundation's **DCE RPC network protocol**, both at the specification and implementation level. The bulk of the implementation effort involved in implementing the COM network protocol is in fact that of implementing the DCE RPC network protocol. COM/DCOM is therefore capable of communicating over all major network and transport protocols, ensuring wide interoperability.

COM+ is the successor to COM/DCOM, incorporating a new generation of technologies (Eddon 1999). COM+ extend COM with respect to **inheritance**, a new **COM Library** (runtime), products like Microsoft Transaction Server, OLE DB, and Microsoft Message Queue as standard COM+ services, and language extensions which makes it easier to build **COM objects** in a variety of programming languages. Although the specification is not fully completed, COM+ has been included in the new Windows 2000 operating system from Microsoft. Backward compatibility is however assured, making any great architectural changes between COM/DCOM and COM+ negligible. This means that DCOM objects using the standard DCOM features will continue to work seamlessly, but certain advanced COM+ features will not be supported. Unfortunately, it appears likely that COM+ will not be made available for non-Microsoft OSs.

4.7 Summary

This chapter highlights COM/DCOM as the other main distributed object infrastructure standard. Designed and developed by Microsoft Corporation, it consists of a specification and an implementation. The specification, in the form of an Object Model, defines *COM objects*, *interfaces*, *COM classes*, *IUnknown*, *delegation*, *aggregation*, *encapsulation*, and *polymorphism*. The implementation, in the form of the COM library, allows transparent communication between the COM objects. The COM library consisting of the service control manager, RPC channels, and utilising the system registry, are currently available on all Microsoft operating systems and most UNIX platforms. The main COM services, including persistent storage, naming, service, uniform data transfer, transaction, query, event, licensing, security, message, and life cycle, are also outlined. Lastly, the interoperability of COM/DCOM and the new COM+ standard is discussed. In the next chapter, meta-modelling is introduced as the semi-formal mechanism for comparing CORBA and COM/DCOM.

Chapter 5 : Meta-Modelling

5.1 Introduction

In the previous chapter, the DCOM object-oriented middleware standard was expounded. In this chapter, CORBA and DCOM are compared. For an accurate and objective dissertation, the comparison between DCOM and CORBA is based on a uniform and unbiased approach. The approach chosen is meta-modelling. Meta-modelling is defined in Brinkkemper (1990) as '*the process of the conceptualisation of a modelling system.*' In utilising meta-modelling for comparing the two object-oriented middleware standards, two distinct meta-models are used, namely the *meta-data* and *meta-process* model. Stam (1995) defines a meta-model as '*a conceptual model of a modelling system.*' A meta-model can therefore be thought of as a conceptual model of a system, providing a single, common, and unambiguous statement of its syntax and semantics.

In the first section of this chapter, a comparative framework is specified, consisting of an **object model**, **infrastructure**, and **services**. The next section defines the two meta-models used for the comparison of the object model and infrastructure of CORBA and DCOM. Additionally, the services provided by CORBA and DCOM are also described by using a summarised service table. The actual comparison is done, based on the meta-data models and service table, in the form of a set of tables in which the similarities and differences of CORBA and DCOM are exhibited. The last section summarises the chapter, providing an overview of meta-modelling, the two meta-models, and the comparison.

5.2 Comparative Framework

The aim of a **comparative framework** is to provide the elements for comparison, and to define **how** it is performed (Brinkkemper 1990). This section therefore outlines the elements used for the comparison between CORBA and DCOM, and how the comparison is done using these elements. By carefully studying the overview of CORBA and DCOM in the previous two chapters, three core elements can be identified. These include the *object model*, *infrastructure*, and *services*.

- **Object model**, in the context of object-oriented middleware, refers to the collection of concepts used to describe objects in a specific object-oriented middleware specification. CORBA and DCOM each has their own object model, forming the basis of each specification.

- The **infrastructure**, defined by the **architecture** of the two object-oriented middleware specifications, allows one object, called the client, to request service from another object, called the server, over a network. This is the most basic function of object-oriented middleware.

- **Services**, also called **object services**, provide a base set of services encapsulated within object-oriented middleware. In other words, object services augment the base functionality of CORBA and DCOM.

The comparison between these elements can be divided into two phases.

- The **first phase** involves the meta-modelling of selected elements of the framework. Firstly, meta-models are used to represent the object models of CORBA and DCOM. The *infrastructure* of CORBA and DCOM are then modelled as meta-models. Because the CORBA and DCOM infrastructure allows an object to request services from a remote object, it implies that specific activities take place in the infrastructure. Meta-process models are used to represent these activities. The services of CORBA and DCOM are then compared, based on a summarised service table, defined later in the chapter.

- Using the same meta-modelling constructs for these two distributed object infrastructures, a uniform and formal representation of CORBA and DCOM can be obtained. Using this representation, the **second phase** of the research compares the distributed object infrastructures, using the comparative framework. The results, given as a set of tables, provide an unbiased and accurate comparison of CORBA and DCOM.

By utilising a comparative framework, this research focuses on key areas of each specification, highlighting their inherent similarities and differences. In the next section, meta-models are introduced.

5.3 Meta-Models

The two meta-models used for meta-modelling the elements of CORBA and DCOM, are the **meta-process model** and the **meta-data model**. The meta-process model describes at a conceptual level the steps or activities of a method or process. The meta-data model, on the other hand, describes the concepts provided by the system or method (Hong, Goor & Brinkkemper 1993). In the following two sections, each model is discussed in more detail, including the graphic representation scheme used.

5.3.1 Meta-Process Model

A meta-process model captures the dynamic aspects of a method or process. For the graphic representation of a meta-process model, the **Task Structure Diagram (TSD),** defined in Hong *et al.* 1993, is used. A *task* or *process*, represented by a rounded rectangle, is the activity (activities) to be followed to transform inputs into a desired product. Tasks can be defined recursively within tasks. This process of decomposition continues until the desired level of detail has been obtained. For example, Task 1 consists of the subtasks Task 1.1 to Task 1.3, illustrated in *Figure 5.1*.

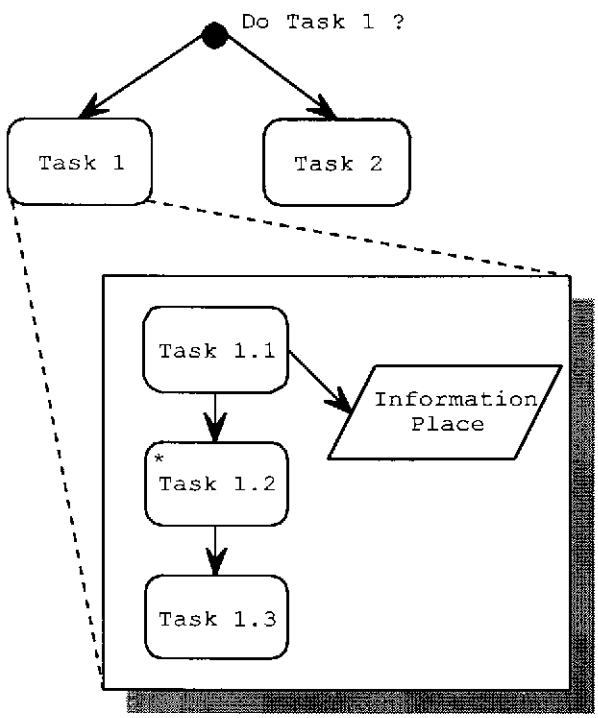


Figure 5.1: Notation for Task Structure Diagrams

[derived from (Hong *et al.* 1993)]

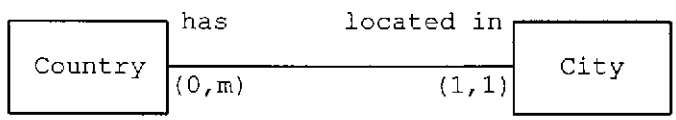
The *products* of tasks are called information places, represented by parallelograms. Each *product* is a deliverable, resulting from the execution of specific tasks. Products can again be used as input to subsequent tasks. When representing decisions, small circles are used. This allows the choice of which task to perform at a specific point. Output dependencies between tasks, are denoted by arrows. Tasks may be performed iteratively, defined by an asterisk (*) in the top left corner of a rounded rectangle, illustrated by Task 1.2 in *Figure 5.1*. This indicates that the task is repeated 1 or more times.

5.3.2 Meta-Data Model

The meta-data model captures the static aspects of a system or method, depicting both the definitions of the entities, and the relations between them (Hong *et al.* 1993). It therefore forms the basis of the conceptual model. For the graphical representation of a meta-data model, the **Extended Entity Relationship (EER)** model as described by Batini, Ceri & Navathe (1992) is used, with some restrictions and extensions as specified in Stam (1995). The main concepts of the EER model include *entities*, *relations*, *cardinality*, *attributes*, and *specialisation*, enunciated in the following paragraphs.

An **entity** represents a collection or set of things in the real world whose members can be identified uniquely. Entities are graphically represented by means of rectangles. Entities are connected to each other by means of **relations**. Normally, relations are drawn as diamonds in EER models, but to reduce complexity and ease of reading, lines are used to graphically represent relations. Relations can be extended by role names, thereby enhancing the perception of the relationship.

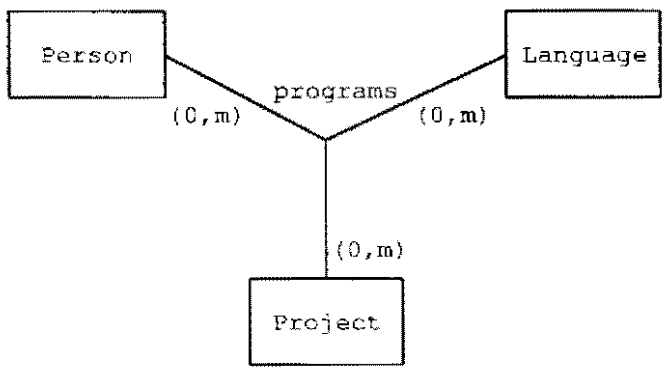
Rules for entities and relations are indicated by **cardinality**, represented by the notation (min,max). For example, a country can have zero or more cities. Conversely, a city can only be located in one country, illustrated in *Example 1*.



Example 1: Cardinality notation

[derived from (Stam 1995) Figure 6.2]

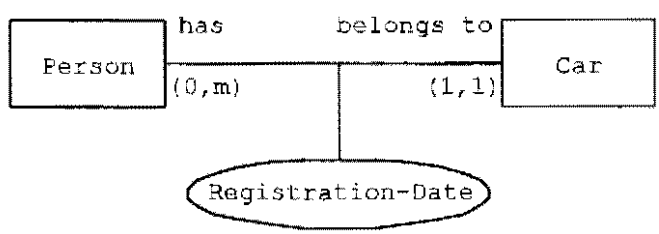
For a ternary relation the notation is as shown in *Example 2*. In the diagram, a person who works as a programmer can use any number of programming languages on any number of projects. A programming language can be used by any number of programmers (persons) on any number of projects. A project can use any number of programmers using any number of programming languages.



Example 2: Ternary relation notation

[derived from (Stam 1995) Figure 6.2]

Attributes are properties of entities and relations, represented as circles or ellipses, containing the name of the attribute. For example, the attribute registration-date of the relation between Person and Car is illustrated in *Example 3*.

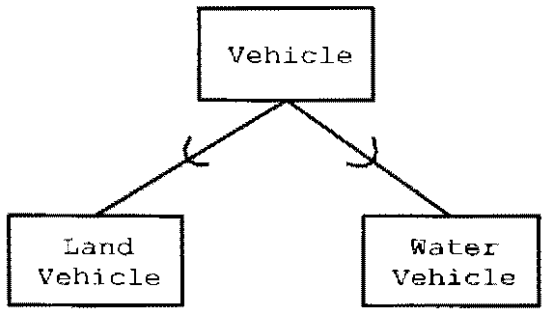


Example 3: Attribute notation

[derived from (Stam 1995) Figure 6.3]

The EER model represents **specialisation** to define a set of subclasses of an entity. For example, the entity Vehicle can be further divided into the subclasses Land Vehicle and Water Vehicle. These subclasses share some common features with the superclass Vehicle. This is represented graphically by means of a subset symbol on the relationship line connecting the subclass to the superclass.

The closed part of the subset symbol always points towards the subclass, as illustrated in *Example 4*.



Example 4: Specialisation

[derived from (Stam 1995) Figure 6.4]

In the following two sections, meta-data and meta-process models are created for two elements in the comparative framework, namely the *object model* and *infrastructure*.

5.4 Meta-Models of the Object Models of CORBA and DCOM

To compare the object models of CORBA and DCOM, meta-modelling is used. In the following two sections, the meta-data models of the CORBA object model and DCOM object model are described. Since the object models of CORBA and DCOM only specify the concepts of each object-oriented middleware standard, no meta-process models are created. Meta-process models are however developed in the modelling of the infrastructure element of the comparative framework.

5.4.1 Meta-Data Model of the CORBA Object Model

The meta-data model of the CORBA object model is illustrated in *Figure 5.2*. In the figure, the most important CORBA object model concepts, namely *object*, *type*, *interface*, *operation*, *attribute*, *request*, and *inheritance*, are modelled, as well as the relationships between them. For each concept a summary is provided, as well as constraints that must be considered. Aspects not explicitly modelled in the figure are also highlighted, namely *encapsulation* and *polymorphism*.

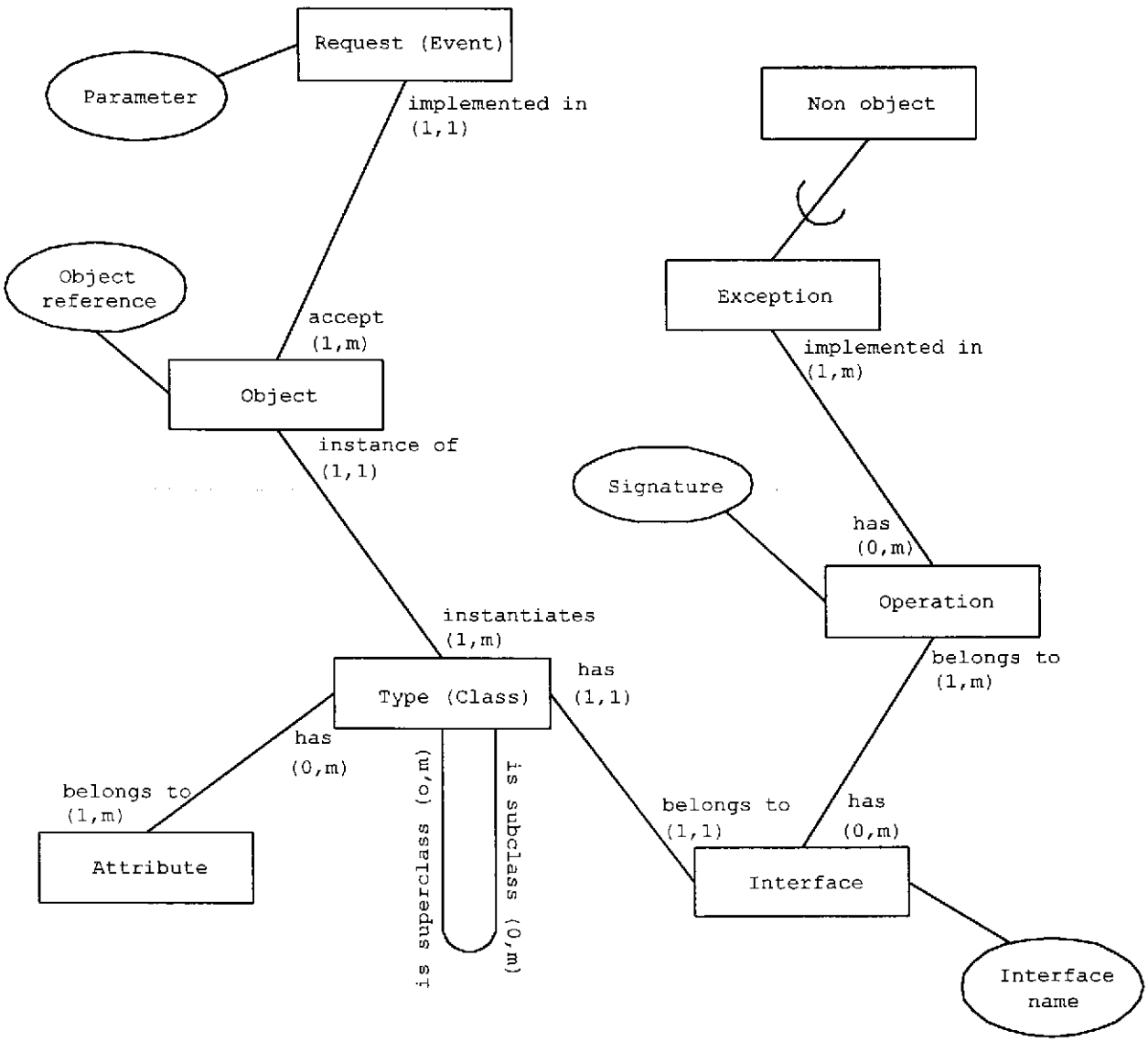


Figure 5.2: Meta-Data Model of the CORBA Object Model

Constraints

- A CORBA **interface** is defined in one, and only one, **type**.
- Each method, called an **operation** in CORBA, must be implemented in one or more **interface**.

Summary of concepts of the CORBA object model

- A CORBA **object** is an identifiable, encapsulated entity that provides services through its interface.
- CORBA defines an **object reference** that identifies an instance of an object uniquely.
- Objects are instances of classes, called **types** in CORBA
- Each **operation** in CORBA has a **signature** that includes an operation name, a set of parameters, and a set of result types.

- A **request** (message), defined as an event, requests services (operations) from a target object. **Parameters** are used to pass data to the object.
- An **interface** is an abstract collection of related operations, uniquely identified by an interface name. It is important to note that the interface of a type consists not only of the operation signatures defined within the type, but also of signatures that are inherited from supertypes.
- An **exception** in CORBA is an indication that an operation request was not performed successfully. Exceptions are normally defined as a *specialised* **non-object** in the CORBA Object Model, containing optional fields for providing information on the causes of abnormal operation termination.
- CORBA supports both single and multiple **inheritance**.

Aspects not represented graphically

Because of inherent limitations to the EER model, encapsulation and polymorphism are not represented graphically. They are however supported in the CORBA object model.

- CORBA supports **encapsulation** by defining an interface for each object.
- CORBA allows a request for a specific operation to be handled differently, depending on the object type on which it is invoked, thereby supporting **polymorphism**.

In the following section, the meta-data model of the DCOM object model is discussed.

5.4.2 Meta-Data Model of the DCOM Object Model

As with the meta-data model of the CORBA object model, certain constraints, concepts, and aspects not represented graphically, must be considered when viewing the meta-data model of the DCOM object model (illustrated in *Figure 5.3*). In the figure, the relationships between the different DCOM object model concepts are represented on a conceptual level. The most important of these include *object*, *server*, *class*, *interface*, *attribute*, *interface pointer*, and *member function*. Aspects not explicitly modelled in the figure include *exceptions*, *encapsulation*, *polymorphism*, and *inheritance*.

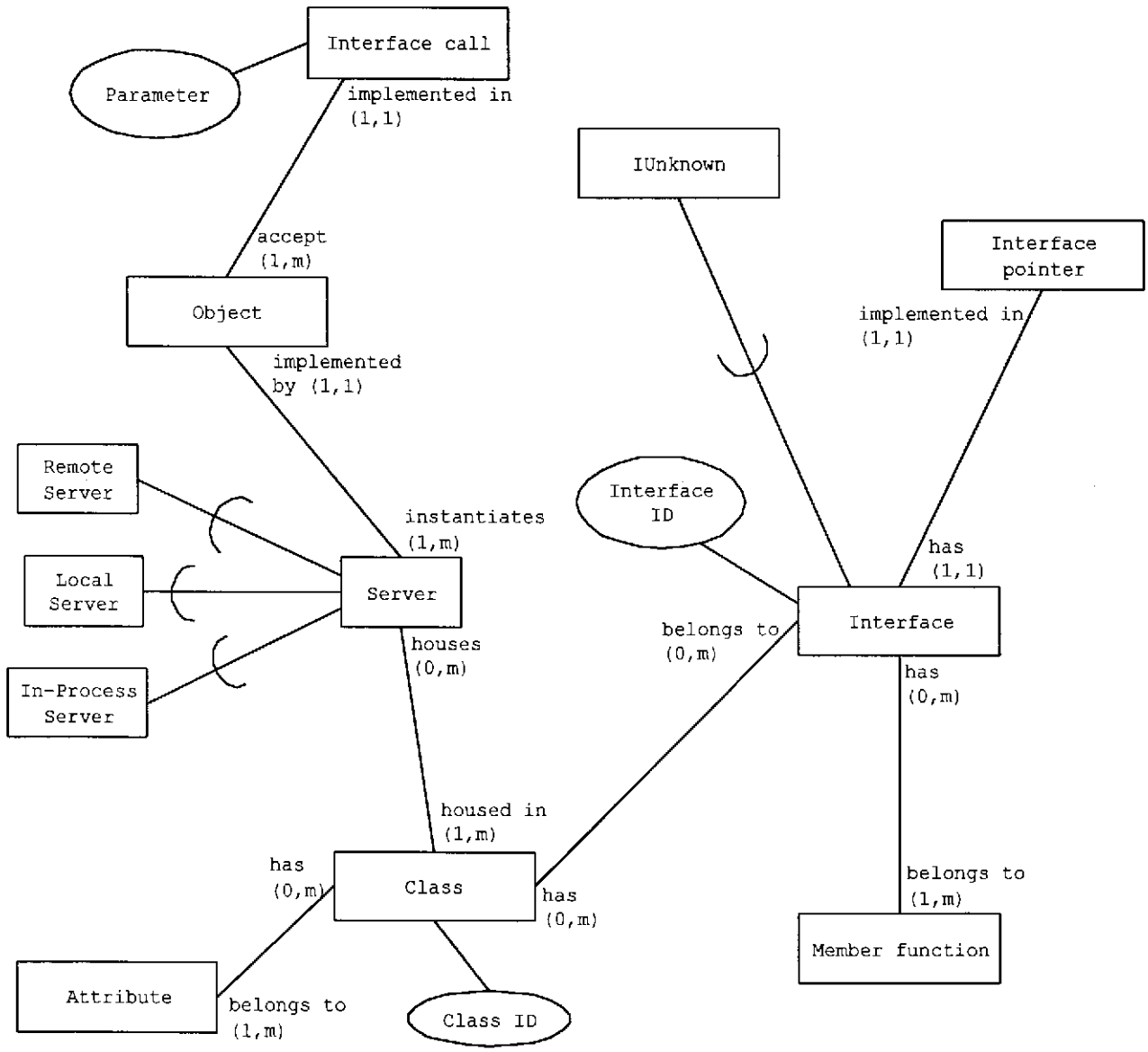


Figure 5.3: Meta-Data Model of the DCOM Object Model

Constraints

- A COM **class** can be housed in more than one COM **server**.
- Each **member function** must be implemented in one or more DCOM **interface**.

Summary of concepts of the DCOM object model

- A COM **object** can support **multiple interfaces**, each representing a different view or behaviour of the object. The association between a COM class and the set of interfaces it supports is purely arbitrary. There implies that there is no established binding of the class identifier to a particular set of interfaces, nor is an instance of a class required to support the same set of interfaces as any other instance of the

class. Instead, a CLSID refers to a particular implementation. The actual set of interfaces supported by an instance of a COM class cannot be known with certainty until execution.

- An **interface,** is an abstract collection of one or more **member functions**, each assigned a *globally unique identifier* called an Interface ID. Each interface implements the *IUnknown* interface, and its three functions *QueryInterface*, *AddRef*, and *Release*. Each interface also has an **interface pointer**, through which it is accessed.
- Messages in DCOM are implemented by means of **interface calls**, specifying the CLSID, IID, and other required parameters.
- A DCOM **class** is a particular implementation of certain set of interfaces. A DCOM **server** houses one or more DCOM classes, each with their own CLSID supplying services to COM clients. Three types of COM servers can be identified, namely **in-process**, **local**, and **remote,** which create object instances of multiple classes.

Aspects not represented graphically

- DCOM does not support the concept of an **object reference**. With DCOM, clients obtain a pointer to an interface (**interface pointer**) and not a pointer to an object with state. DCOM clients can therefore not reconnect to an object instance with the same state at a later time. The closest analogy is the *Moniker*, which provides a persistent naming mechanism.
- DCOM does not support **exceptions**. COM interfaces can only use a specific *return value*, called HRESULT, to indicate status and error information.
- DCOM does not support **inheritance**. Instead, DCOM uses **multiple interface** support to implement *aggregation* and *delegation*, enabling a type of object reuse.

Because of inherent limitations to the EER model, encapsulation and polymorphism are not represented graphically. They are however supported in the DCOM object model.

- DCOM supports **encapsulation** by defining one or more interfaces for each object.
- DCOM supports **polymorphism**.

The meta-data models illustrated in the last two sections, together with the constraints, summary of concepts, and aspects not represented graphically, are used in *section 5.7.2* to actually compare the object models of CORBA and DCOM. In the next section, the infrastructures of CORBA and DCOM are described by utilising meta-modelling.

5.5 Meta-Modelling the Infrastructures of CORBA and DCOM

In the following two sections, meta-process models and meta-data models are developed for the infrastructures of CORBA and DCOM. The meta-data models present the concepts of the DCOM and CORBA infrastructures, while the meta-process models illustrate the activities in the utilisation of each infrastructure.

5.5.1 Meta-Process Models and Meta-Data Model for the Infrastructure of CORBA

The meta-process models for the infrastructure of CORBA are given in *Appendix A*, *Figures 1 to 3*. Three levels of meta-process models are represented, namely level 0, level 1, and level 2. The activities or tasks defined in each level becomes progressively more detailed. As in Hong *et al.* 1993, only the resulting products of activities have been denoted. It is also important to note that although the activities appear sequentially, iteration can take place. Because CORBA supports both static and dynamic invocation, it is also necessary to define this distinction in the meta-process models. The main reason is that the activities differ between a static and dynamic invocation.

The meta-data model of the CORBA infrastructure is given in *Appendix A*, *Figure 4*. The following assumptions and observations are important when viewing the meta-data model.

Assumptions

- A **stub** can be linked to more than one **client**.
- A **skeleton** can be linked to more than one **object implementation**.

Observations

- A **client** can use more than one **stub**.
- A client can use one and only one **dynamic invocation interface**.
- Both the **skeletons** and dynamic skeleton interface use the **object adapter.**
- An **object implementation** can be linked to more than one **skeleton**.
- The **interface repository** is used by the **dynamic invocation interface**.
- The **implementation repository** is used by the **dynamic skeleton interface**.

In the following section, the meta-models of the DCOM infrastructure are outlined.

5.5.2 Meta-Process Models and Meta-Data Model for the Infrastructure of DCOM

The meta-process models for the infrastructure of DCOM are given in *Appendix B, Figures 1 to 3.* As for the meta-process models of the CORBA infrastructure, three levels of meta-process models are drawn, namely level 0, level 1, and level 2. Because DCOM also supports static and dynamic invocation, it is necessary to define this distinction in the meta-process models. The meta-data model of the CORBA infrastructure is given in *Appendix B, Figure 4.* The following assumptions and observations must be considered when viewing the meta-data model.

Assumptions

- A **proxy** is linked to one **client**.
- A **stub** is linked to one **object implementation**.

Observations

- The **COM Library** consists of the **Type Library** and **Service Control Manager**.
- The **COM Library** uses the **registry** to find the path for server.
- The **COM library** uses the **RPC runtime** for communication.
- A **class factory** has zero or more **object implementations**.

In the following section, the services provided by CORBA and DCOM are summarised for comparison.

5.6 Comparing the Services of CORBA and DCOM

Because CORBA and DCOM view their services differently, it makes a comparison inherently difficult. Microsoft enumerates their COM-related services as *security, life cycle management, type information, naming, database access, data transfer, registry* and *asynchronous communications.* CORBA, in contrast, defines more than fifteen well-designed services in their CORBAservices specification, discussed in *Chapter 3.* The definition of a summarised, one-on-one service table is therefore crucial, namely *Table 5.1.* In the table, the services of CORBA and DCOM are summarised, with a short description of the support provided by each object-oriented middleware.

Union of CORBA and DCOM Services	CORBA	DCOM
Type information	Part of ORB	Provided as a DCOM service.
Registry	Part of ORB.	Provided as a DCOM service.

Union of CORBA and DCOM Services	CORBA	DCOM
Data transfer	Part of ORB.	Provided by Uniform Data Transfer service.
Asynchronous communications	Currently not supported. Will be part of CORBA 3.0.	Provided by the Microsoft Message Queue product.
Life cycle	Supported, defining operations to create, copy, move and delete objects.	Supported by AddRef and Release functions of IUnknown interface.
Persistence	Provides a single interface for storing objects on a variety of storage servers.	Provided by DCOM's Persistence Storage service.
Event	Allows objects to register or unregister interest in specific events.	Provided by connectable objects, defining incoming and outgoing interfaces.
Transactions	Allows 2PC co-ordination among objects of either flat or nested transactions.	Available in the Microsoft Transaction Server product include services like transactional guarantees, concurrency control (thread synchronisation), database session management, and security
Concurrency	Provides a lock manager that can obtain locks on behalf of either transactions or threads.	Not provided.
Relationships	Allows dynamic association between objects that know nothing of each other.	Not provided.
Externalisation	Provides a standard way of getting data into and out of an object, using a stream-like mechanism.	Not provided.
Security	Incorporates a mechanism for authentication, based on encrypted passwords.	Utilise the OS security mechanisms.
Time	Provides management of global time for distributed objects.	Not provided.
Query	Provides query operations for objects.	Provided by the Microsoft OLE DB product.

Union of CORBA and DCOM Services	CORBA	DCOM
Licensing	Provides operations for metering the use of objects to ensure fair compensation for their use.	Implemented by the IClassFactory2 interface, which enforce licensing at object creation time.
Properties	Allows association of properties to objects.	Provided in DCOM.
Trader	Provides a matchmaking service between clients seeking services and objects offering services.	Not supported.
Version	Allows the support of different versions of objects.	Not provided, must create new interface ID for each new interface.
Collection	Supports the creation and manipulation of collections of objects.	Not supported.

Table 5.1: The Summarised Service Table of CORBA and DCOM

The summarised service table, defined in this section, is used in the following section to compare the object services provided by CORBA and DCOM.

5.7 Comparing CORBA and DCOM

The actual comparison of CORBA and DCOM, based on the comparative framework, is presented in the following three sections. The result is given in a set of three tables in which the similarities and differences of CORBA and DCOM are exhibited. First the object models of CORBA and DCOM are compared in a table, followed by the infrastructure of each object-oriented middleware. The last section compares the services provided by CORBA and DCOM.

5.7.1 Comparing the Object Models of CORBA and DCOM

To compare meta-data models, the concepts provided in each model are compared. This is done by listing the union of all concepts in a table, and comparing them based on symbols, described in *Table 5.2*.

Symbol	Description
*	An asterisk denotes that the specific concept is supported by CORBA or DCOM.
<name>	A name denotes that the specific concept is supported by CORBA or DCOM, but has a different name.
	A blank represents the absence of a corresponding concept in CORBA or DCOM

Table 5.2: Legend for comparing the concepts of CORBA and DCOM

The union approach has been adopted to provide a fair and equal comparison between the CORBA and DCOM object models. Another approach could entail taking the CORBA object model as the base, and comparing it with the DCOM object model. This however would entail prejudice towards the DCOM object model. The comparison between the CORBA and DCOM object models is shown in *Table 5.3*.

Object Model	CORBA	DCOM
Object	*	*
Single Interface	*	*
Multiple Interfaces		*
Interface Pointer		*
Object Reference	*	
Class	Type	*
Method	Operation	Member function
Message	Request	Interface call
Exception	*	
Encapsulation	*	*
Polymorphism	*	*
Single Inheritance	*	
Multiple Inheritance	*	

Table 5.3: Comparing the Object Model concepts of CORBA and DCOM

The major difference between the two object models is that CORBA follows the **classical object model**, and extends it to distributed objects, while DCOM is more an extension of the **binary standard** of interfaces, used in C++ and OLE. CORBA objects are therefore truly object-oriented, supporting single and multiple inheritance, encapsulation, and polymorphism. COM/DCOM objects are not truly object-oriented since no inheritance is supported. A COM class cannot be extended via inheritance. COM/DCOM however allows the creation of complex classes by multiple interfaces, which enables **delegation** and **aggregation**. Together delegation and aggregation provides Microsoft's version of *inheritance*.

5.7.2 Comparing the Infrastructure of CORBA and DCOM

The activities of CORBA are compared against the activities of DCOM. This approach is chosen to provide a direct comparison between the two object-oriented middleware infrastructures. The actual comparison of activities is done by listing the activities of the DCOM infrastructure, and comparing them to the CORBA infrastructure activities, based on the symbols listed in *Table 5.4*.

Symbol	Description
=	The activity of CORBA is equivalent to the activity of DCOM.
<	The activity of CORBA is contained in the activity of DCOM.
-	The activity of CORBA is not part of the activity of DCOM.
>	The activity of CORBA consists of more than the activity of DCOM.
><	The activities of both CORBA and DCOM have an overlapping and non-overlapping part.

Table 5.4: Legend for comparing the activities of CORBA and DCOM

In *Appendix D Table 1*, the activities of DCOM is listed, showing how each corresponding activity of CORBA compares with it. It can be seen that DCOM and CORBA have much in common. Most activities in DCOM have a similar, although not identical partner in CORBA. The main difference is that DCOM implements a **ClassFactory** object to create object instances, while this is not the case with CORBA. DCOM also does not support **exceptions**, which is provided in the dynamic invocation interface and stubs of CORBA. DCOM and CORBA however both support static and dynamic invocation.

In *Table 5.5*, the CORBA and DCOM infrastructure concepts are compared. This is accomplished by comparing the union of the concepts, based on the symbols described in *Table 5.2*. Like the comparison between the CORBA and DCOM object models, the union approach has been adopted to provide a fair and equal comparison between the CORBA and DCOM infrastructures.

Infrastructure	CORBA	DCOM
Client	*	*
Object Implementation	*	*
Class factory		*
Stub	*	Proxy
Skeleton	*	Stub
Object Request Broker	*	COM Library
Dynamic Skeleton Interface	*	IDispatch Interface
Dynamic Invocation Interface	*	IDispatch Interface
Interface Repository	*	Type Library
Implementation Repository	*	Registry
Object Adapter	*	
Service Control Manager		*
Object Request Broker Interface	*	
RPC Runtime		*

Table 5.5: Comparing the Infrastructure concepts of CORBA and DCOM

It can be seen that the infrastructure of CORBA and DCOM are also very similar. Each infrastructure defines a client, which requests services from an object implementation. Some naming differences do however occur. CORBA calls its *client stub* a **stub**, while DCOM calls it a **proxy**. The *server stub* is called a **skeleton** in CORBA, but DCOM calls it a **stub**. Also important to note is that the **interface repository** in CORBA is the same as the **type library** in DCOM. Similarly, the **implementation repository** in CORBA is equivalent to the **registry** in DCOM. On the lower levels the concepts of CORBA and DCOM do differ, for example, CORBA does not implement the concepts of a service control manager or RPC runtime, these concepts are part of the Object Request Broker.

5.7.3 Comparing the Services of CORBA and DCOM

In conclusion, the services of CORBA and DCOM are compared. This is done based on *Table 5.1*, the summarised service table, defined in *section 5.6*.

The CORBA and DCOM services listed in *Table 5.1* is compared in *Table 5.6*, using the symbols employed in comparing the meta-process models of CORBA and DCOM, defined in *Table 5.2.*

Services	CORBA	DCOM
Type information	*	*
Registry	*	*
Data transfer	*	*
Asynchronous communications		
Life cycle	*	*
Naming	*	*
Persistence	*	*
Event	*	*
Transactions	*	
Concurrency	*	
Relationships	*	
Externalisation	*	
Security	*	*
Time	*	
Query	*	
Licensing	*	*
Properties	*	*
Trader	*	
Version	*	
Collection	*	

Table 5.6: Comparing the Service of CORBA and DCOM

From the table it is evident that there are many overlapping services, for example, *type information*, *registry*, *data transfer*, *life cycle*, *naming*, *persistence*, *event*, *security*, *licensing*, and *properties*. It can also be seen that the CORBA specification supports more services than DCOM. This is mainly due to the formal specification approach followed by the OMG, included in the *Object Management Architecture Reference Model*, allowing the adoption of a wider range of services. Microsoft prefers to develop new products to extend the services of DCOM, for example Microsoft Transaction Server (transaction service), Microsoft Message Queue (asynchronous communications), and OLE DB (query services). These services, which

are not provided in DCOM, are standard services in CORBA, mainly due to the input received by the OMG from researchers and organisations involved in the formal specification process.

Another major difference between CORBA and DCOM services is that DCOM services are tightly integrated with all Microsoft operating systems. Microsoft's strategy is to make their OSs the platforms of choice. It is not clear what subset of these services will be made available on other OS platforms. However, it seems that Microsoft will always support its OS platforms as the *de facto* standard for COM/DCOM, hoping to increase its market share even further in the OS sector.

5.8 Summary

This chapter introduces meta-modelling, the approach chosen to compare CORBA and DCOM. Meta-modelling is based on the creation of meta-models for specific elements, defined in the comparative framework. In this research the *object model*, *infrastructure*, and *object services* of CORBA and DCOM are chosen as the elements of the comparative framework. For the object models of CORBA and DCOM, meta-data models are created. Similarly, meta-data and meta-process models are created for the infrastructures of CORBA and DCOM. For the services of DCOM and CORBA a summarised service table is used, listing the union of services provided by each, and how they compare. These meta-models, which form a uniform and unbiased representation, are then used in the actual comparison of CORBA and DCOM, listed as a set of tables.

From the comparison it can be seen that both object-oriented middleware specifications have much in common. There are however also major differences between them, mainly due to the way each specification was created and extended. DCOM was developed from OLE and COM, setting it in a fixed framework. CORBA was developed by consensus between multiple parties, making it more flexible and representative. In the next chapter, an overview of the research is provided, drawing crucial conclusions on the future of software development and object-oriented middleware.

Chapter 6 : Summary and Conclusion

6.1 Introduction

Although the semi-formal comparison between CORBA and DCOM is of great benefit for distributed object computing, the intent of this study is not to identify a best choice. Instead, this research used the meta-modelling technique to build a formal representation of CORBA and DCOM, allowing an **accurate**, **unbiased**, and **extensive comparison**.

Both CORBA and DCOM are still evolving, increasing the possibility of further conversion in specific areas. Each standard also represents tremendous effort and experience in providing a distributed object infrastructure. Specific characteristics of CORBA and DCOM must however be considered. Because Microsoft Corporation both developed the DCOM specification, and its implementation, it is predominately utilised in a Microsoft OS environment. CORBA, in comparison, is not linked to a specific OS or platform, resulting in improved heterogeneous environment support. This is mainly due to the fact that many different vendors have implemented the CORBA specification, resulting in a wide range of CORBA products.

In the following sections, a summary of the dissertation is given, concentrating on middleware, CORBA, DCOM, and the comparison by means of meta-modelling. Future research and other comparative points are also enunciated.

6.2 Middleware

Middleware is the software that enables the elements of software applications to communicate over networks, despite underlying differences in protocols and OSs. *Chapter 2* provides a short introduction to middleware, concentrating on the five different categories of middleware that can currently be defined. These include *RPC based middleware*, *message oriented middleware*, *database middleware*, *TP monitors*, and **object-oriented middleware**. It is however the last category, namely object-oriented middleware, that forms the focus of this dissertation. This is due to the fact that CORBA and DCOM are classified as OOM. In the following two sections, each of these object-oriented middleware standards are further reviewed.

6.3 CORBA

CORBA, outlined in *Chapter 3*, is currently one of the most popular object-oriented middleware standards. Developed by the OMG, it is based on the OMG Object Management Architecture, comprising the **Core**

Object Model and **Reference Model**. The *Core Object Model* defines the standard's basic concepts including *objects*, *types*, *operations*, *interfaces*, *subtyping*, and *non-object types*. The *Reference Model* is an architectural framework for the standardisation of interfaces used by applications consisting of five elements, namely the *object request broker*, *object services (CORBAservices)*, *common facilities (CORBAfacilities)*, *domain interfaces (CORBAdomains)*, and *application objects*.

The **object request broker**, which forms the architecture of CORBA, consists of the *client*, *object implementation*, *ORB core*, *stub*, *dynamic invocation interface*, *skeleton*, *dynamic skeleton interface*, *Object Adapter*, *ORB interface*, *interface repository*, and *implementation repository*. By utilising this architecture, CORBA allows a client to make a request to an object implementation situated on any other platform. *CORBAservices* incorporate a collection of services required to support the basic functions for using and implementing objects, namely *life cycle*, *naming*, *persistence*, *event*, *transactions*, *concurrency*, *relationship*, *externalisation*, *time*, *query*, *licensing*, *properties*, *trader*, *version* and *collection*.

CORBAfacilities represents a collection of services that many applications may share, but which are not as fundamental as *CORBAservices*. *CORBAdomains* represents vertical areas that provide functionality of direct interest to end-users in particular application *domains*, for example financial, telecommunications, and manufacturing. Also defined by the OMG is an interface definition language, mainly used to describe the interfaces that a client call and object implementation provide. Currently, the OMG provide IDL mappings for C, C++, SmallTalk, Cobol, Ada, and Java.

6.3.1 The Future of CORBA

Companies like IBM, Netscape, and Oracle, are actively deploying technologies that are CORBA compliant. IBM has introduced Component Broker, its new CORBA technology, which is 100% Java compliant. **Component Broker** is being marketed as IBM's strategic CORBA platform, a complete solution for customers needing to connect multiple back-end systems to distributed object applications. Netscape has built the **Java ORB runtime** into its Navigator browser. Oracle is using CORBA technology in an attempt to make its vision of distributed computing a reality. The Oracle vision, in its most basic form, is to have lightweight clients download Java applets that communicate to server applications and databases by means of CORBA/IIOP. JavaSoft, the creator of the very successful Java language, is also actively adding support for CORBA, allowing Java objects to communicate by means of CORBA/IIOP. With large software vendors like IBM, Netscape, Oracle, and JavaSoft backing CORBA, and with CORBA 3.0 adding more support for the Internet and components, CORBA will play a significant role in the future of distributed object computing.

6.4 DCOM

DCOM, discussed in *Chapter 4*, is the de facto other OOM standard developed by Microsoft Corporation. It is an extension of COM, providing COM objects with the ability to communicate in a heterogeneous environment. The development of COM/DCOM has its roots in OLE. OLE is a *compound document framework*, supporting the linking and embedding of objects created in one application, called a *server*, into a document in another application, called a *container*. The OLE architecture is a collection of two higher-level services, namely *OLE Compound Documents* and *OLE Automation*, built upon COM.

The COM/DCOM specification, defined in the **COM Object Model**, specify the different elements of COM, including *COM objects and interfaces*, *COM clients and servers*, *COM classes*, *IUnknown*, *multiple interfaces*, *encapsulation*, and *polymorphism*. In addition to being a specification, COM/DCOM is also an implementation, contained in the **COM Library**. The architecture of DCOM is built around the COM Library, providing the most basic communication services. Other elements forming part of the architecture include the *client*, *object implementation*, *proxy*, *stub*, *service control manager*, and *registry*. By utilising this architecture, DCOM allows a client to make a request to an object implementation situated on any other platform.

COM Services incorporate a collection of services required to support the basic functions for using and implementing COM objects. These include two services previously classified as part of OLE, namely *persistent storage* and *uniform data transfer*, and other services like *naming*, *event*, *licensing*, *security* and *life cycle*. Also developed by Microsoft is the Microsoft IDL, based on simple extensions to the OSF's DCE interface definition language. MIDL is mainly used to define a custom interface, describing its data types and methods. From the interface definition, the MIDL compiler generates the source code necessary to build the *proxy* and *stub* for the COM object, as well as the *Type Library*. MIDL is however a tool of convenience and not central to COM's interoperability. It intrinsically saves the developer from manually creating proxies, stubs, and Type Libraries by hand.

6.4.1 The Future of DCOM

The heavy reliance of DCOM on Microsoft OSs is a major stumbling block. This prompted Microsoft to port DCOM to other OSs. A Microsoft solutions provider, Software AG, has developed DCOM ports to the major UNIX OSs. Even more DCOM-porting initiatives are underway at **Digital Equipment Corporation (DEC)**, HP, Siemens Nixdorf, NCR Corporation, and Sybase. The bottom line is that DCOM will eventually have multi-platform support, giving the DCOM architecture more credibility as a tool for building distributed object applications.

Another complaint of DCOM is that the development effort is awkward and complex. Fortunately, Microsoft's Visual J++, Microsoft's implementation of the Java language, removes some of the complexity by making DCOM objects resemble Java classes. Interestingly enough, Java is better suited for DCOM development than other languages such as C++, Visual Basic, and Delphi, which is mainly due to the Visual J++ tools provided by Microsoft. Although DCOM reliance on Microsoft OSs is viewed as negative, it currently provides an immediate access to more than 250 million Microsoft OS platforms. The projected $4 billion dollar market in 2001 for third-party components based on DCOM, is also impressive. COM+, the successor DCOM, will solve many of the problems currently experienced with DCOM. DCOM will, like CORBA, play a significant role in the future of distributed object computing.

6.5 Meta-Modelling: CORBA versus DCOM

Meta-modelling, discussed in *Chapter 5*, is the approach chosen to compare CORBA and DCOM. Consisting of the creation of meta-models for specific elements of each object-oriented middleware standard, namely the *object model* and *infrastructure*, it allows for an unbiased comparison. Two types of meta-models used in the uniform representation of these elements are meta-process and meta-data models. Meta-process models describe the activities or processes, while meta-data models describe the concepts.

The **object models** of CORBA and DCOM are compared by only using meta-data models. In this comparison, specific similarities and differences can also be seen. Both CORBA and DCOM support the object-oriented concepts of *object*, *single interface*, *class*, *method*, *message*, *encapsulation* and *polymorphism*. CORBA however do not support *multiple interfaces*, a concept used by DCOM to support *delegation* and *aggregation*, enabling a type of object reuse. DCOM again does not provide for *object references (OIDs)*, *exceptions*, and both *single* and *multiple inheritance*. The DCOM object model can therefore not be classified as a **classical object model** as is the case for the CORBA object model, mainly as a result of its evolution from OLE.

For the **infrastructure** of CORBA and DCOM, both meta-data and meta-process models are created. This allows the comparison of the concepts provided in each infrastructure, and the activities of each. It is important to note that both CORBA and DCOM have many concepts and activities in common when considering their infrastructures, but specific differences also occur. DCOM utilises class factories to create object instances, while CORBA does not. DCOM also breaks its ORB into sub-components like COM Library, service control manager, and RPC runtime, while the CORBA infrastructure only implements an ORB. It can therefore be seen that the DCOM infrastructure is more complex, breaking the activities followed into more detailed steps.

The **services** of CORBA and DCOM are compared by means of a summarised table. This provides a clear representation of the services each object-oriented middleware supports. DCOM only provides services like *security*, *life cycle management*, *type information*, *naming*, *database access*, *data transfer*, *registry* and *asynchronous communications*. CORBA, in contrast, defines sixteen well-designed services, namely *life cycle*, *naming*, *persistence*, *event notification*, *transactions*, *concurrency*, *relationships*, *externalisation*, *security*, *time*, *query*, *licensing*, *properties*, *trader*, *version* and *collection*.

Although CORBA and DCOM both support crucial services like life cycle and naming, CORBA does have an advantage over DCOM in this area. The DCOM object services are tightly integrated with the operating system, supporting Microsoft's strategy to make its OSs the platforms of choice. In contrast, CORBA services are supported on various OSs and platforms. It can also be seen that the CORBA specification defines more services than DCOM. This is mainly due to the formal specification approach followed by the OMG, included in the *Object Management Architecture Reference Model*, allowing the adoption of a wider range of services. Another difference between CORBA and DCOM is support for reusability, specifically for services. Since CORBAservices (services of CORBA) form part of the OMA Reference Model, reusability is supported. This is not the case for DCOM, mainly because of the way Microsoft has implemented DCOM services as separate products, for example the Microsoft Transaction Service developed to support transactions in DCOM. CORBA however defines new services for extending the CORBA specification, not new products.

By utilising meta-modelling to compare the object model, infrastructure, and services of CORBA and DCOM, specific focus is given to key areas of CORBA and DCOM. This enables a clear and concise comparison, adding to the research in the area of distributed object computing.

6.6 Future Research

Distributed object computing, which is based on distributed object infrastructures, promises to change the way software is developed and maintained. Most distributed objects are developed by software companies, and shipped as independent software components (distributed objects). These components are purchased by component integrators and incorporated in standard containers such as compound document frameworks, discussed in *Chapter 4*. As a result, developers can create distributed object applications by simply dragging and dropping components onto containers, configuring them as required. Applications development therefore becomes a paste, layout, and configuration job, requiring very little or no programming. This is the **future of software development**, changing the majority of application programmers into component configurers. When this new software paradigm is combined with the Internet, the impact on software systems could be even more profound. Further research under consideration includes how the Internet can utilise CORBA and COM/DCOM/COM+, specifically concentrating on

component support (Siegel 1999). This is especially crucial when observing the global drive towards Internet software applications and E-Commerce.

6.7 Conclusion

Choosing between CORBA and DCOM was not the objective of this research. Each object-oriented middleware standard represents tremendous effort and experience in providing a distributed object infrastructure. Instead, the main **contribution** of this research is the use of the meta-modelling technique to build a formal representation of CORBA and DCOM, and comparing CORBA and DCOM based on their representation. This approach enables the performance of an **accurate**, **unbiased**, and **extensive comparison**. In this way, errors of misunderstanding or misinterpretation can be detected, and therefore avoided in the comparison process. Secondly, the research result provides information system professionals an extensive overview of CORBA and DCOM, assisting in the evaluation and study of these two distributed object infrastructures. Furthermore, the result is a valuable information resource for organisations that are planning to implement distributed object computing. The research therefore deliberately avoided the identification of a best choice, which would have required a rating of CORBA and DCOM based on current properties. Both CORBA and DCOM have not reached their mature stage, increasing the possibility of further improvement and conversion in the future.

References

(Baker 1997) Baker, S. 1997. CORBA Disributed Objects Using Orbix. Harlow, England: Addison-Wesley: 1-67.

(Batini, Cari & Navathe 1992) Batini, C., Ceri, S. & Navathe, S. 1992. Conceptual Database Design: An entity-relationship approach. Redwood City, Calif.:The Benjamin/Cummings Publishing Company Inc.

(Bouzeghoub, Gardarin & Valduriez 1997) Bouzeghoub, M., Gardarin, P. & Valduriez, P. 1997. Object Technology Concepts and Methods. London: Addison-Wesley.

(Brinkkemper 1990) Brinkkemper, S. 1990. Formalisation of Information Systems Modelling. Amsterdam: Thesis Publishers.

(Brockschmidt 1993) Brockschmidt, K. 1993. Inside OLE. Redmond, Wash.: Microsoft Press.

(Carr 1997) Carr, D. 1997. CORBA and DCOM: How each Works. *Web Week*, 3(7):1-28.

(Chappell 1996) Chappell, D. 1996. Understanding ActiveX and OLE. Redmond, Wash.: Microsoft Press.

(COM 1995) Microsoft. 1995. The Component Object Model Specification. *(http://www.microsoft.com/oledev/olecom/title.htm)*. Redmond, Wash.: Microsoft Press.

(CORBA 1995) Object Management Group. 1995. The Common Object Request Broker: Architecture and Specification Revision 2.0. *(http://www.omg.org/corba/corbiiop.htm)*. Framingham, Mass.

(Douglas & Loosley 1997) Douglas, F. & Loosley, C. 1997. High-Performance Client/Server. New York: John Wiley.

(Eddon 1999) Eddon, G. 1999. COM+: The Evolution of Component Services, *Computer*, 32(7):104-106.

(Emerald Chung, Huang, Yajnik, Liang, Shih, & Wang 1997) Emerald Chung, P., Huang, Y., Yajnik, S., Liang, D., Shih, J., Wang, Y. 1997. DCOM and CORBA Side by Side, Step by Step, and Layer by Layer. *(http://www.cs.wustl.edu/~schmidt/submit/paper.html)*. Bell Laboratories. New York: Murray Hill.

(Gopalan 1998) Gopalan, S. 1998. A Detailed Comparison of CORBA, DCOM, and Java/RMI. *Object Magazine*. 5(23):1-14.

(Harmon & Morrissey 1996) Harmon, P. & Morrissey, W. 1996. The Object Technology Casebook – Lessons From Award-Winning Business Applications. New York: John Wiley.

(Hong, Goor & Brinkkemper 1993) Hong, S. Goor, van den G. Brinkkemper, S. 1993. A Formal Approach to the Comparison of Object-Oriented Analysis and Design Methodologies. Proceedings of the 26th Hawaii International Conference on System Sciences. IV:689-698.

(H7 1997) H7. 1997. Object Model Feature Matrix Report - Doc. No. X3H7-93-007v12b. National Committee for Information Technology Standards.

(Jones 1995) Jones, M. P. 1995. What Every Programmer Should Know About OO Design. London: Dorset House.

(Linthicum 1997) Linthicum, D. 1997. David Linthicum's Guide to Client/Server and Intranet Development. New York: John Wiley.

(Montgomery 1997) Montgomery, J. 1997. Distributed Objects – Special Report. *Byte Magazine*. 4(7):23-35.

(Mowbray & Ruh 1997) Mowbray, T. & Ruh, W. 1997. Inside CORBA Distributed Object Standards and Applications. Menlo Park, Calif.: Addison-Wesley.

(Orfali, Harkey & Edwards 1997)	Orfali, R., Harkey, D. & Edwards, J. 1997. Instant CORBA. New York: John Wiley.
(Orfali, Harkey & Edwards 1996a)	Orfali, R., Harkey, D. & Edwards, J. 1996a. The Essential Client/Server Survival Guide. 2nd Edition. New York: John Wiley.
(Orfali, Harley & Edwards 1996b)	Orfali, R., Harley, D. & Edwards, J. 1996b. The Essential Distributed Objects Survival Guide. New York: John Wiley.
(Rauch 1996)	Rauch, W. 1996. Open Systems Engineering - How to Plan and Develop Client/Server Systems. New York: John Wiley.
(Rogerson 1997)	Rogerson, D. 1997. Inside COM. Redmond. Wash.: Microsoft Press.
(Session, Vogel & Kim 1998)	Session, R., Vogel, A. & Kim, E. 1999. COM vs. CORBA. *Dr. Dobbs Journal*. 8(12):1-17.
(Siegel 1999)	Siegel, J. 1999. A Preview of CORBA 3. *Computer*, 32(5):114-118.
(Stam 1995)	Stam, A. 1995. Development of a Flexible CASE tool. Master Thesis of Design Methodolgy Research Group. Department of Computer Science. University of Twente.
(Tkach & Puttick 1996)	Tkach, D. & Puttick, R. 1996. Object Technology in Application Development. London: Addison-Wesley.
(The Oxford Dictionary of Computing 1997)	University of Oxford. 1997. A Dictionary of Computing. 4th Edition. Oxford: Oxford University Press.
(Tallman & Bradford Kain 1998)	Tallman, O. & Bradford Kain, J. 1998. COM versus CORBA: A Decision Framework. *Distributed Computing*. 4(3).

(Watson, Soley & Bradley 1997) Watson, A., Soley, R. & Bradley, M. 1997. Comparing ActiveX and CORBA/IIOP. *(http://www.omg.org/news/activex.htm)*. Object Management Group, Framingham, Mass.

Appendix A – Meta-Process Models for the Infrastructure of CORBA

In this appendix, the meta-process models for the infrastructure of CORBA are listed.

Meta-Process Models

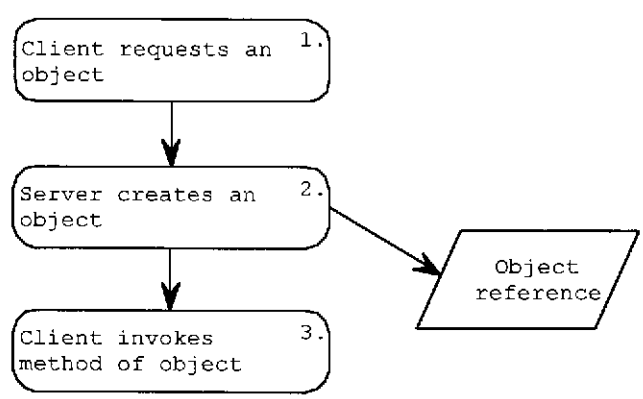


Figure 1: Meta-Process Model for the Infrastructure of CORBA, level 0

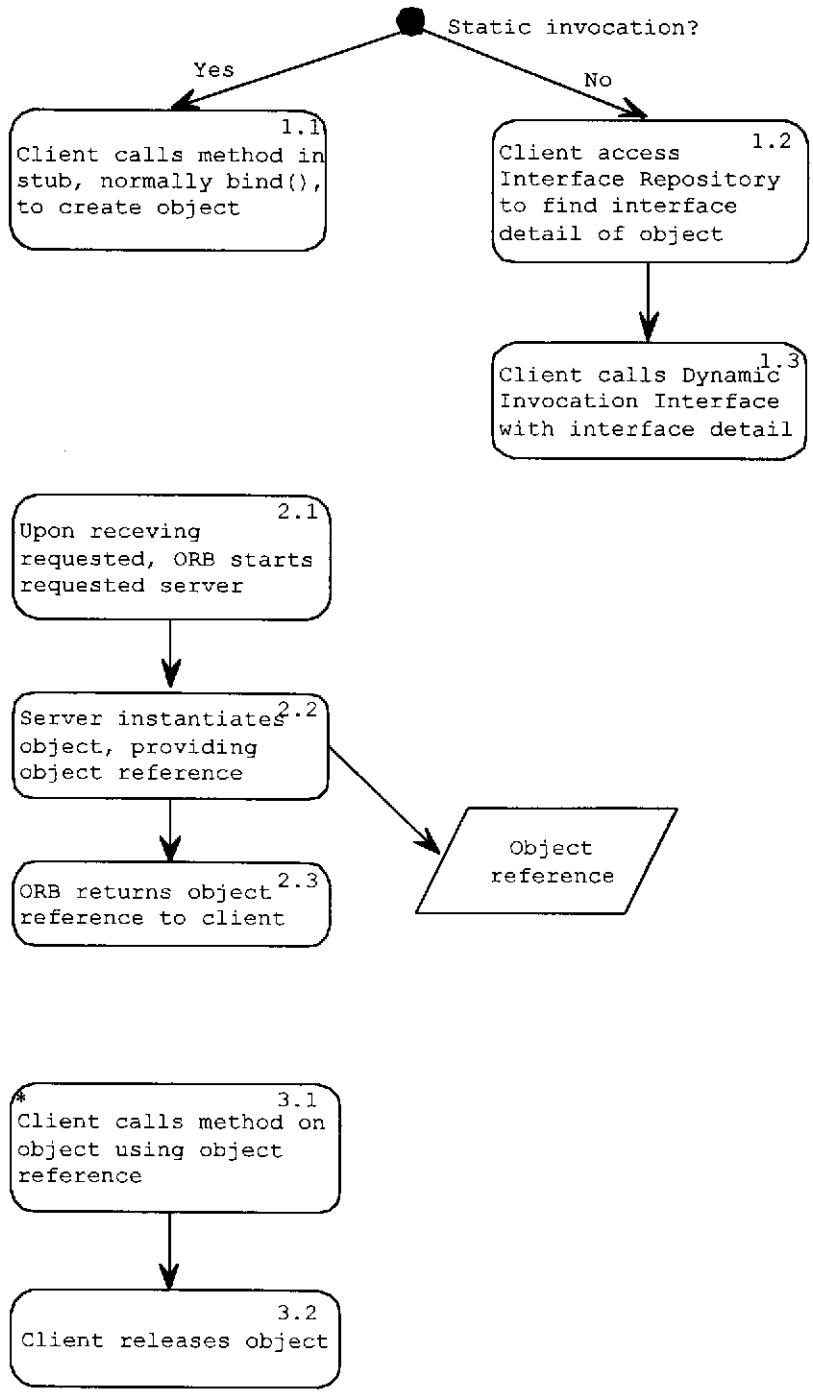


Figure 2: Meta-Process Model for the Infrastructure of CORBA, level 1

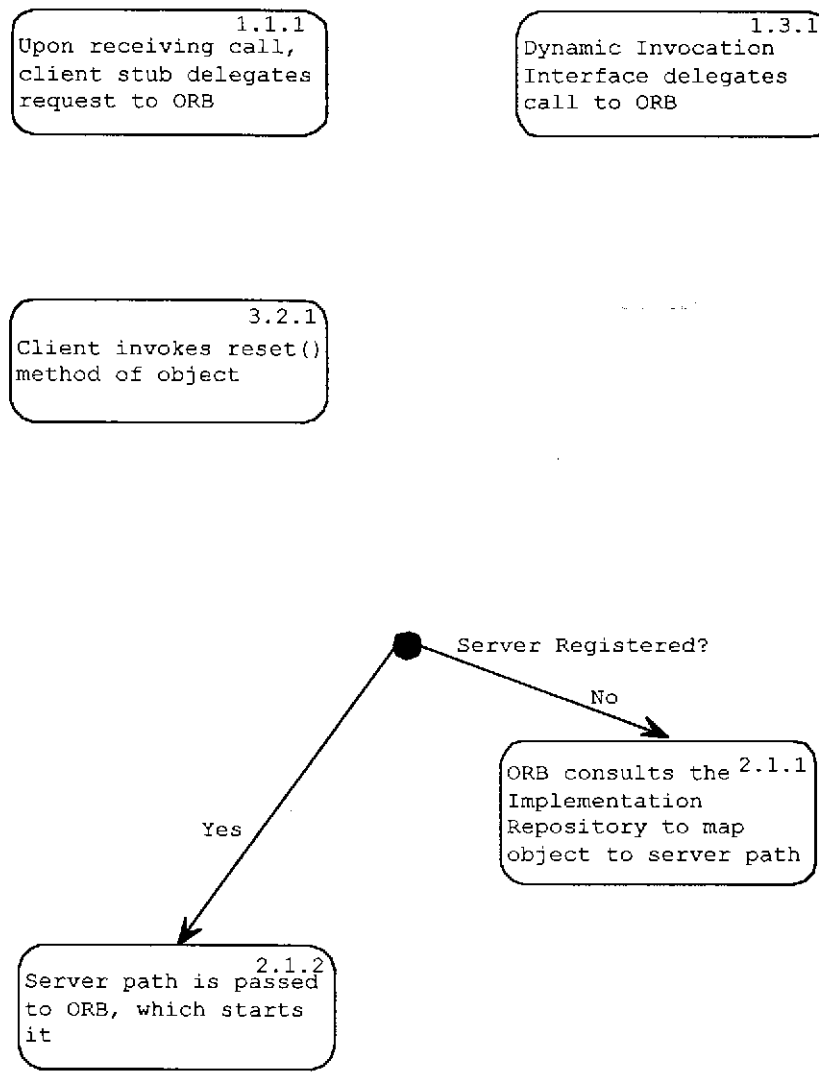


Figure 3(a): Meta-Process Model for the Infrastructure of CORBA, level 2

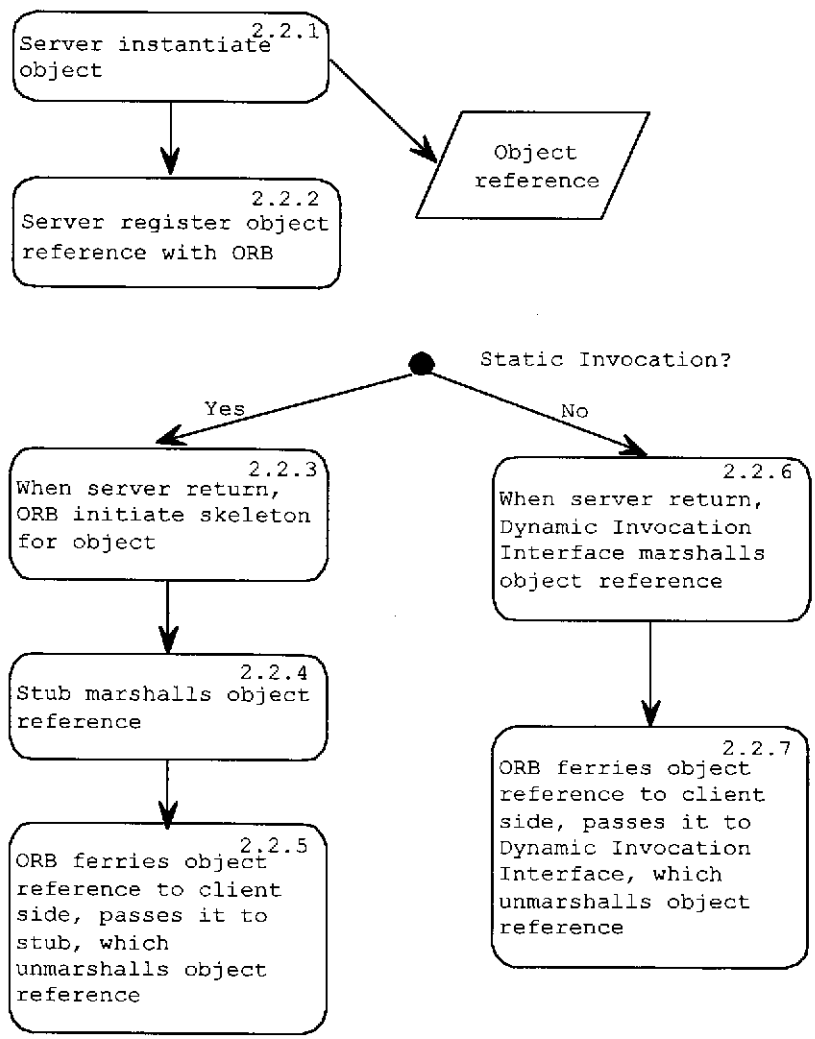


Figure 3(b): Meta-Process Model for the Infrastructure of CORBA, level 2

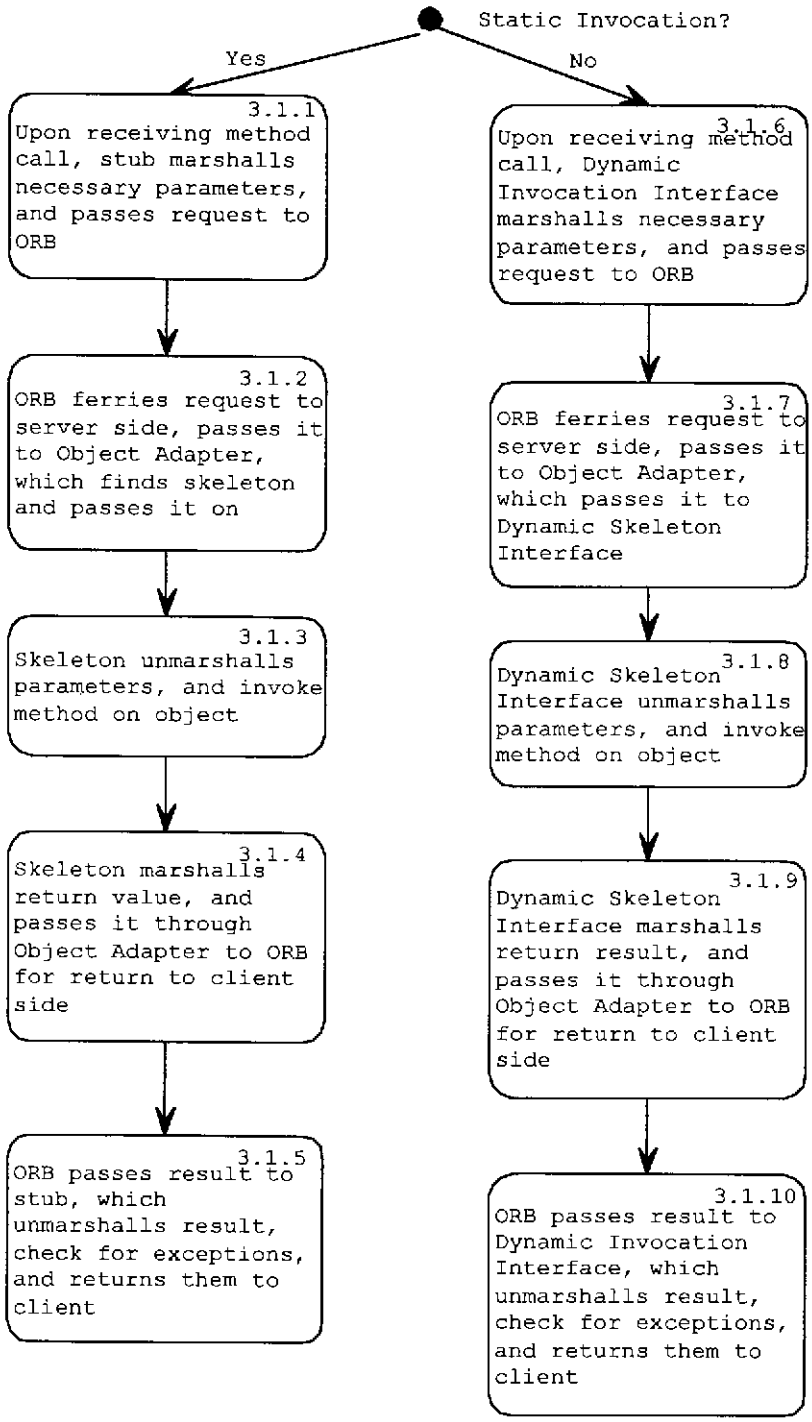


Figure 3(c): Meta-Process Model for the Infrastructure of CORBA, level 2

Appendix B – Meta-Process Models for the Infrastructure of DCOM

In this appendix, the meta-process models for the infrastructure of DCOM are listed.

Meta-Process Models

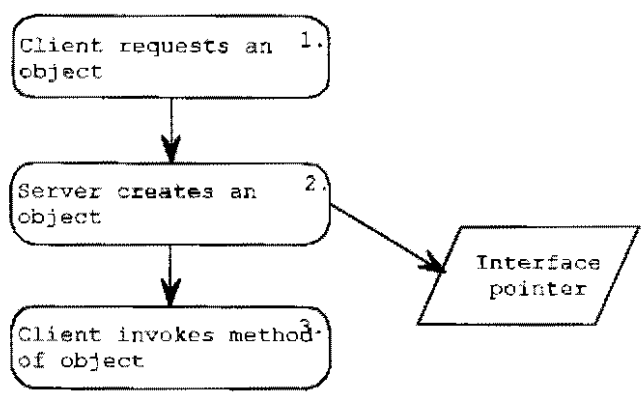


Figure 1: Meta-Process Model for the Infrastructure of DCOM, level 0

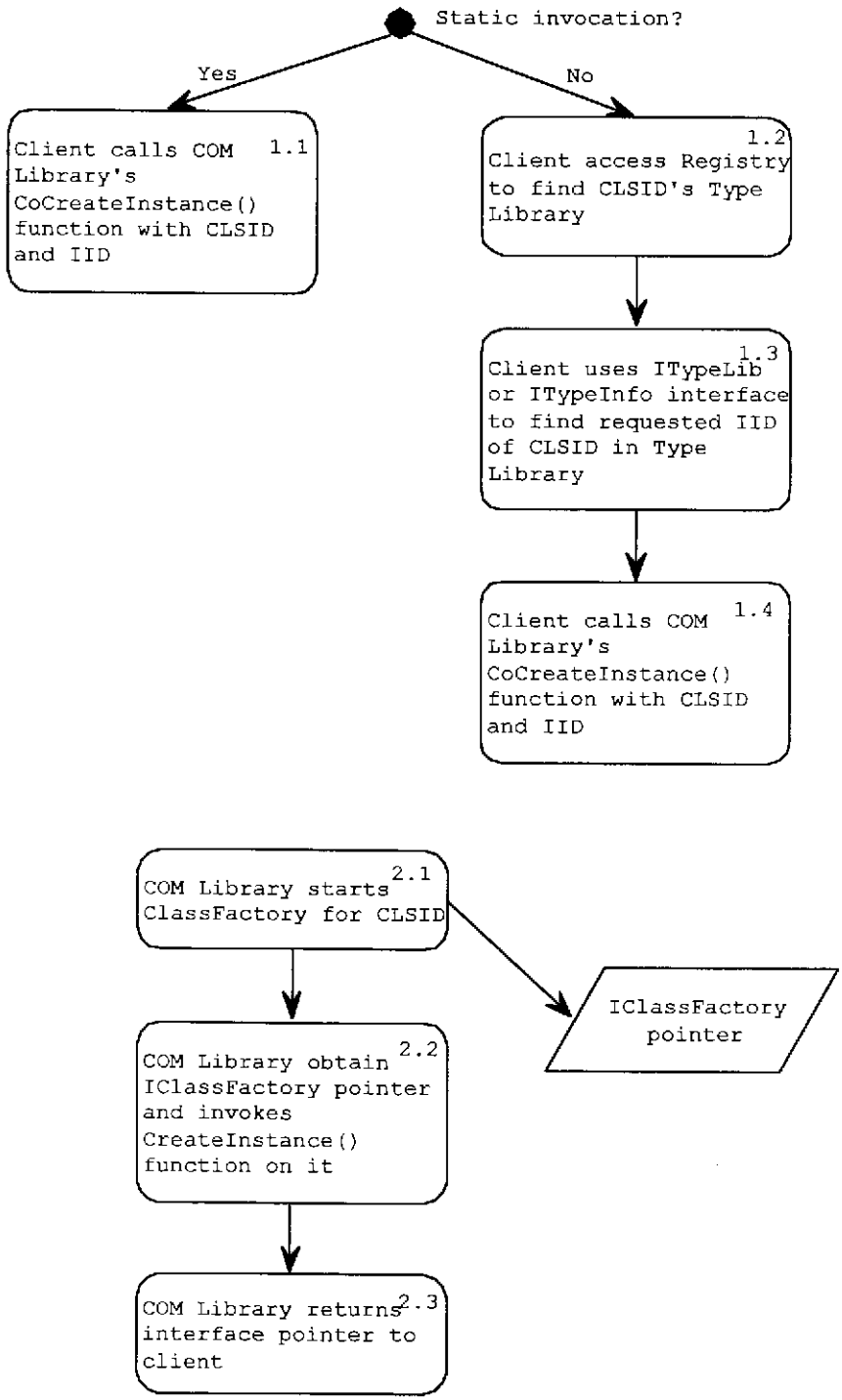


Figure 2(a): Meta-Process Model for the Infrastructure of DCOM, level 1

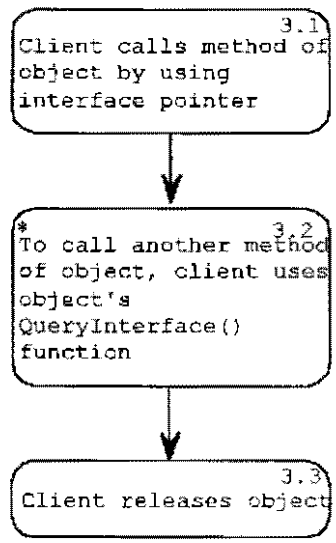


Figure 2(b): Meta-Process Model for the Infrastructure of DCOM, level 1

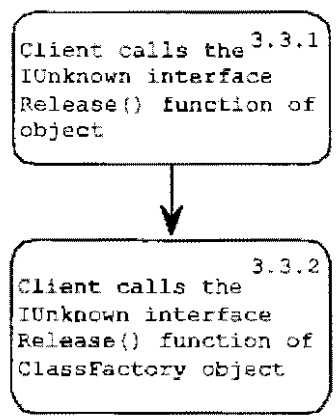


Figure 3(a): Meta-Process Model for the Infrastructure of DCOM, level 2

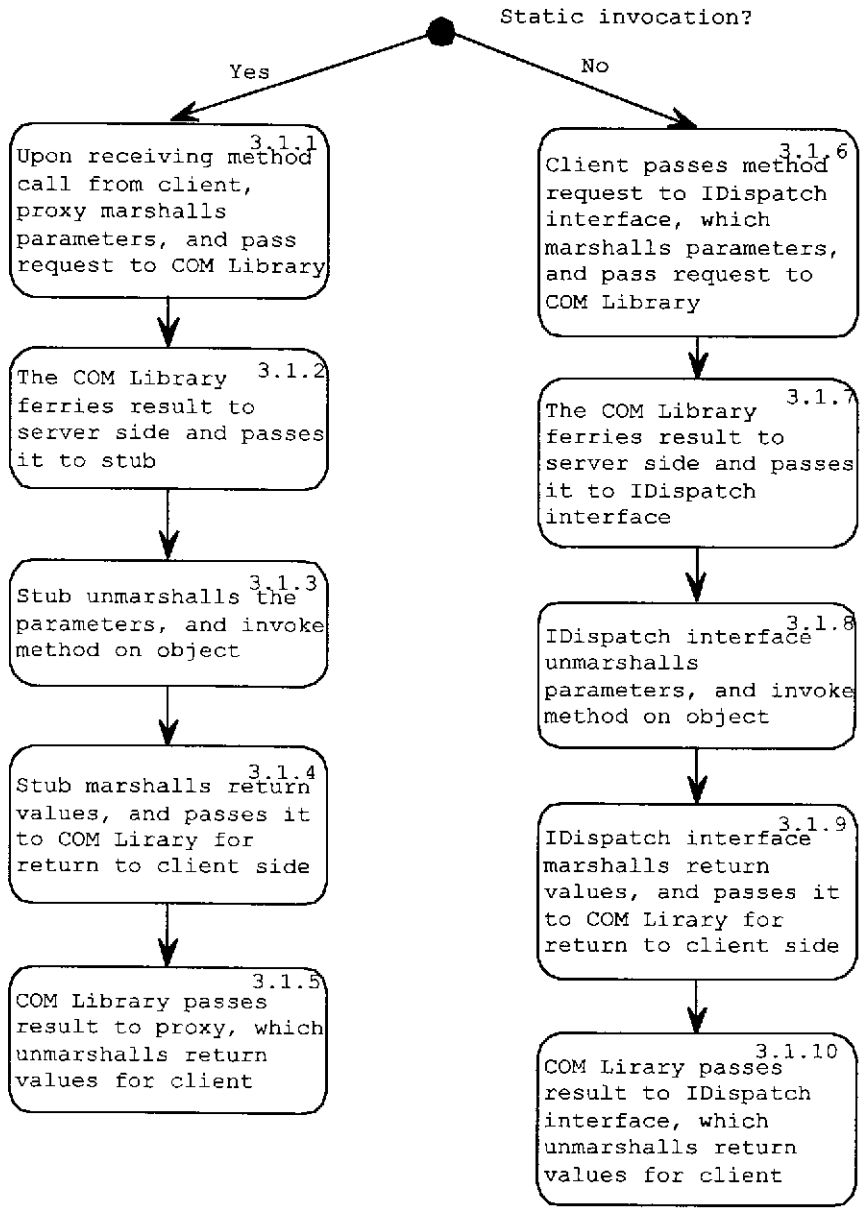


Figure 3(b): Meta-Process Model for the Infrastructure of DCOM, level 2

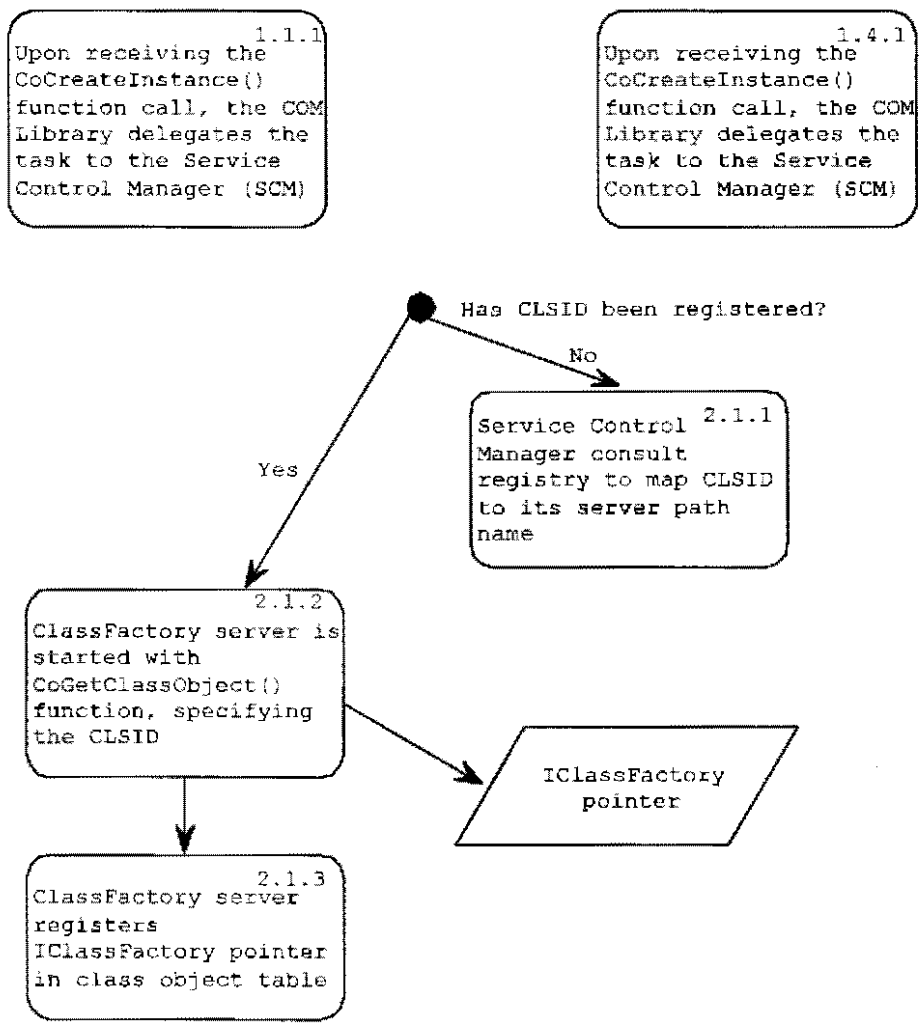


Figure 3(c): Meta-Process Model for the Infrastructure of DCOM, level 2

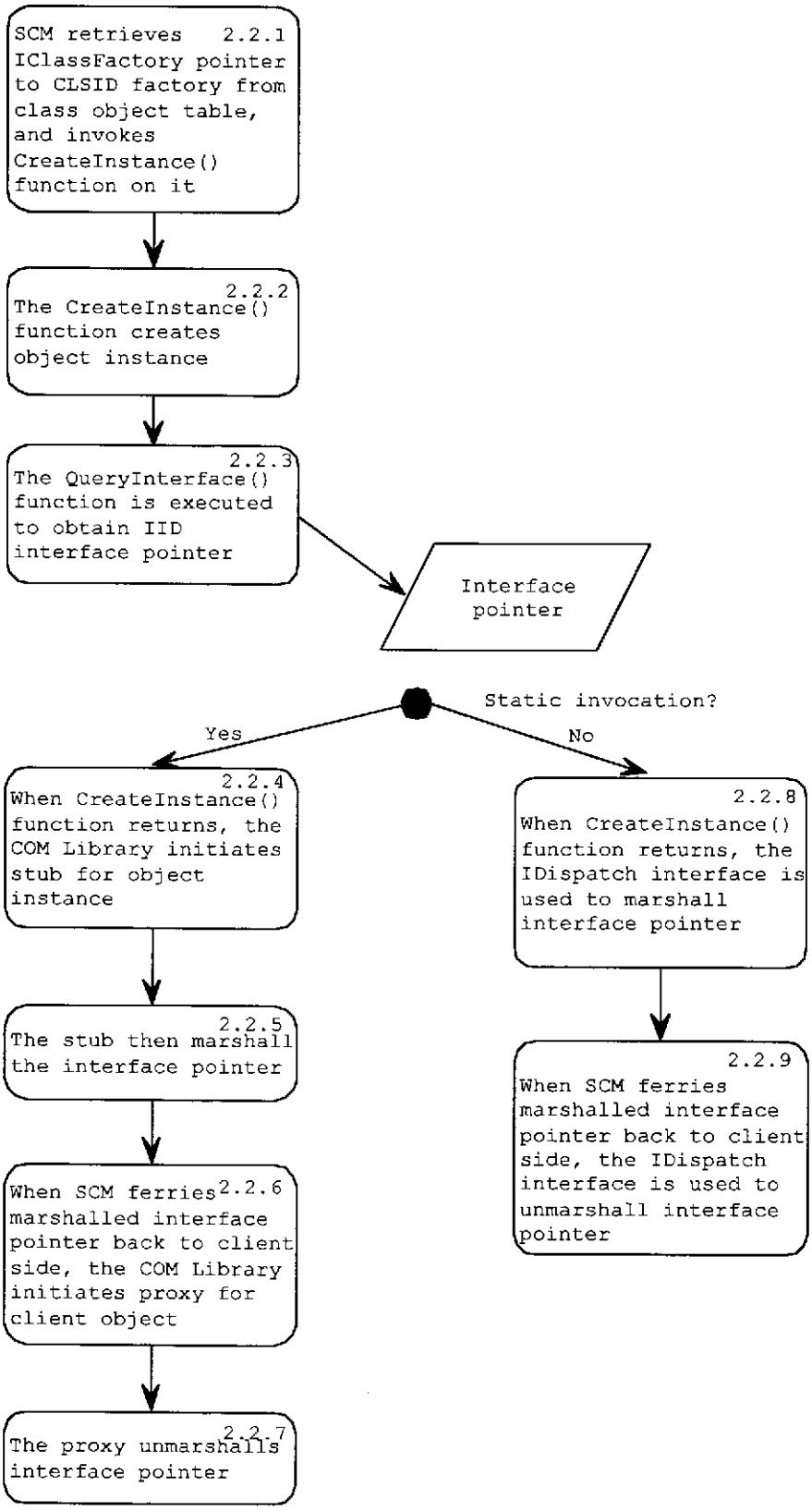


Figure 3(d): Meta-Process Model for the Infrastructure of DCOM, level 2

Appendix C – Comparing the Meta-Process Models for the DCOM and CORBA Infrastructure

In this appendix, the meta-process models of the DCOM infrastructure are compared with the meta-process models of the CORBA infrastructure.

Activity of DCOM	Activity of CORBA	
1. Client request an object	=	1
Static invocation = 'Yes'		
1.1 Client calls COM Library's CoCreateInstance() function with CLSID and IID	<	1.1
1.1.1 Upon receiving CoCreateInstance() function call, the COM Library delegates the task to the Service Control Manager (SCM)	<	1.1.1
Static invocation = 'No'		
1.2 Client access Registry to find CLSID's Type Library	=	1.2
1.3 Client uses ITypeLib or ITypeInfo interface to find requested IID of CLSID in Type Library	-	
1.4 Client calls COM Library's CoCreateInstance() function with CLSID and IID	<	1.3
1.4.1 Upon receiving CoCreateInstance() function call, the COM Library delegates the task to the Service Control Manager (SCM)	<	1.3.1
2. Server creates an object	=	2
2.1 COM Library starts ClassFactory for CLSID	<	2.1
CLSID has not been registered		
2.1.1 Service Control Manager consult registry to map CLSID to its server path name	=	2.1.1

Activity of DCOM	Activity of CORBA	
CLSID has been registered		
2.1.2 ClassFactory server is started with CoGetClassObject() function, specifying the CLSID	<	2.1.2
2.1.3 ClassFactory server registers IClassFactory pointer in class object table	<	2.2.2
2.2 COM Library obtain IClassFactory pointer and invokes CreateInstance() on it	<	2.2
2.2.1 Service Control Manager retrieves IClassFactory pointer to CLSID factory from class object table, and invokes CreateInstance() function on it	-	
2.2.2 The CreateInstance() function creates object instance	<	2.2.1
2.2.3 The QueryInterface() function is executed to obtain IID interface pointer	-	
Static invocation = 'Yes'		
2.2.4 When CreateInstance() function returns, the COM Library initiates stub for object instance	<	2.2.3
2.2.5 The stub the marshalls interface pointer	=	2.2.4
2.2.6 When SCM ferries marshalled interface pointer back to client side, the COM Library initiates proxy for client object	-	
2.2.7 The proxy unmarshalls interface pointer	>	2.2.5
Static invocation = 'No'		
2.2.8 When CreateInstance() function returns, the IDispatch interface is used to marshall interface pointer	=	2.2.6
2.2.9 When SCM ferries marshalled interface pointer back to client side, the IDispatch interface is used to unmarshall interface pointer	=	2.2.7
2.3 COM Library returns interface pointer to client	=	2.3
3. Client invokes method of object	=	3

Activity of DCOM	Activity of CORBA	
3.1 Client calls method of object by using interface pointer	<	3.1
Static invocation = 'Yes'		
3.1.1 Upon receiving method call from client, proxy marshalls parameters, and pass request to COM Library	=	3.1.1
3.1.2 The COM Library ferries result to server side and passes it to stub	>	3.1.2
3.1.3 Stub unmarshalls parameters, and invoke method on object	=	3.1.3
3.1.4 Stub marhalls return values, and passes it to COM Library for return to client side	>	3.1.4
3.1.5 COM Library passes result to proxy, which unmarshalls return values for client	>	3.1.5
Static invocation = 'No'		
3.1.6 Client client passes method request to IDispatch interface, which marshalls parameters, and passes request to COM Library	=	3.1.6
3.1.7 The COM Library ferries result to server side and passes it to IDispatch interface	>	3.1.7
3.1.8 IDispatch interface unmarhalls the parameters, and invoke method on object	=	3.1.8
3.1.9 IDispatch interface marhalls return values, and passes it to COM Library for return to client side	>	3.1.9
3.1.10 COM Library passes result to IDispatch interface, which unmarshalls return values for client	>	3.1.10
3.2 To call another method of object, client uses object's QueryInterface function	-	
3.3 Client releases object	=	3.2
3.3.1 Client calls the IUknown interface Release() function of object	=	3.2.1
3.3.1 Client calls the IUknown interface Release() function of ClassFactory object	-	

Table 1: Comparison of the activities of CORBA with the activities of DCOM