A DISTRIBUTED COMPUTING ENVIRONMENT (DCE) BASED OBJECT REQUEST BROKER

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE,

FACULTY OF SCIENCE

AT THE UNIVERSITY OF CAPE TOWN

IN FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

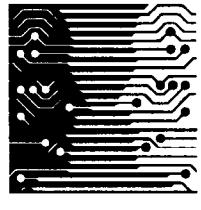
By

Anique van der Vlugt

December 1995

Supervised by

Prof. KJ MacGregor



Abstract

Object oriented technology has moved beyond being a tool for design and programming and is now being used to implement enterprise wide computer systems. Also, there has been a move from centralised mainframe systems to distributed computing due to the advent of more powerful workstations and faster, more reliable networks. The integration of object oriented technology and distributed computing is becoming a generally accepted method for implementing networked computer solutions.

The purpose of the research presented in this thesis is to investigate how the evolving object oriented technologies can build upon the current distributed computing technology by using there underlying infrastructure and then to implement a CORBA compliant distributed Object Request Broker. This involves the design and implementation of a compiler which maps CORBA objects to DCE remote procedure calls. Our objective is to investigate the operation of a distributed object implementation and in particular the performance which can be achieved by a DCE-based Object Request Broker which is CORBA compliant.

Acknowledgments

I would like to thank everyone who assisted me in the preparation of this thesis, especially the following people without whose help my task would have been immeasurably more difficult :

- I would like to thank my supervisor Prof. Ken MacGregor for introducing me to the field of distributed object computing, as well as his financial support and guidance;
- The staff and students at the Computer Science department, University of Cape Town;
- I am grateful to Prof. Henk Goosen for his endless patience in teaching me C++ and our computer laboratory assistants - Sandi Donno and Aleks Strez - for their technical assistance;
- The Foundation for Research and Development for their financial support,
- And my family and friends whose support helped me through the last agonizing months.

Thank you sincerely.

List of Abbreviations

API	Applications Programming Interface
BOA	Basic Object Adaptor
CDR	Common Data Representation
CIOP	Common Inter-ORB Protocol
CORBA	Common Object Request Broker Architecture
DCE	Distributed Computing Environment
DII	Dynamic Invocation Interface
ESIOP	Environment Specific Inter-ORB Protocol
IDL	Interface Definition Language
IIOP	Internet Inter-ORB Protocol
IOR	Interoperable Object Reference
IR	Interface Repository
NDR	Network Data Representation
OLE	Microsoft's Object Linking and Embedding Mechanism
OMA	Object Management Architecture
OMG	Object Management Group
ORB	Object Request Broker
OSF	Open Software Foundation
RPC	Remote Procedure Call

Glossary

- **Basic Object Adapter (BOA)**
  The BOA is defined in the CORBA 1.2 specification to be used for most ORB objects with conventional implementations.

- **Context**
  A **Context** object contains a list of "properties" that represent information about an application process's environment. Each **Context** property consists of a <name,string_value> pair, and is used by application programs or methods much like the *environment variables* commonly found in operating systems.

- **CORBA Interface Definition Language (IDL)**
  In CORBA, an object is an instance of a class. The class provides the definition of the interface to that object (and all other objects of that class). Therefore, we use CORBA IDL to define a class interface which are the methods and their parameters, return types, exceptions and contexts.

- **DCE Interface Definition Language (IDL)**
  In distributed applications, the DCE IDL file contains definitions the client and server share, and a list of all the procedures offered by the server.

- **Dynamic Invocation Interface (DII)**
  The interface definition is used to dynamically build requests on remote objects by querying the Interface Repository.

- **Dynamic Skeleton Interface (DSI)**
  The DSI is a way to deliver requests from an ORB to an object implementation that does not have compile-time knowledges of the type of object it is implementing.

- **Environment**
  Each request requires an **Environment** parameter which represents a memory location where exception information can be returned by the object of the memory invocation to the client application.

- **General Inter-ORB protocal (GIOP)**
  The standard interoperability for network ORBs.

- **ImplementationDef**
  An **ImplementationDef** object is used to describe an object's implementation. Typically, the **ImplementationDef** describes the program that implements an object's server, how the program is activated, and so on. **ImplementationDef** objects are stored in the *Implementation Repository*.

- **Implementation Repository**
  A database used to store the implementation definitions of the object implementations.

- **InterfaceDef**
  An **InterfaceDef** object is used to describe an IDL interface in a manner that can be queried and manipulated at runtime when building requests dynamically, for example. **InterfaceDef** objects are stored in the Interface Repository.

- **Interface Repository**
  The database used to provide persistent storage of objects representing the major elements of interface definitions. Creation and maintenance of the IR is based on the information supplied in the IDL source file.

- **Internet Inter-ORB-Protocol (IIOP)**
  A specific mapping of a GIOP which runs under TCP/IP connections.

- **Object Adapter (OA)**
  The primary interface a server implementation uses to access ORB functions; in particular it defines the mechanisms that a server uses to interact with the ORB. This includes server activation/deactivation, dispatching of methods, and authentication of the principal making the call.

- **Object**
  An entity that has state (its data values) and behaviour (its methods).

- **Object Adapter (OA)**
  A CORBA interface which defines generic object adapter (OA) methods that a server can use to register itself and its objects with an object request broker (ORB).

- **Object Implementation**
  An **object implementation** provides the actual data for the object instance and code for the object's methods. The implementation also interacts with the ORB to establish its identity and to create new objects.

- **Object Request Broker (ORB)** is a mechanism for controlling the interaction of distributed objects

- **Principal**
  A **Principal** object identifies the principal (user) on whose behalf a request is being performed.

- **Request**
  A **Request** object represents a specific request on an object, constructed at runtime. The **Request** object contains the target object reference, operation (method) name, a list of input and output arguments. A **Request** can be invoked synchronously (wait for the response), asynchronously (initiate the call, and later, get the response), or as a **oneway** call (no response expected).

- **Server**
  See *Object Implementation*.

- **ServerRequest**
  The **ServerRequest** structure stores the explicit state of a request for the DSI which is analogous to the `Request` structure in the DII.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Scope of the Thesis

The integration of different paradigms of computing presents many interesting problems in computer science. The traditional functional approach which has been used for many years has offered stability in the computer industry. The more modern object orientated approach has now come to the fore. The question that arises is how can these two approaches be married?

This thesis presents an attempt to investigate how an object orientated distributed computing environment that is CORBA compliant can be layered on top of a functional distributed computing approach as illustrated by DCE.

The body of the work presents an investigation of the distributed computing environment and its interface definition language. The mapping of this IDL to the CORBA IDL is also discussed. A CORBA compliant object request broker has been implemented using the underlying features found in DCE IDL.

The efficiency of this object request broker is then compared to that of available object request broker to assess the overhead of integrating the two technologies. It was found that the locally developed object request broker compared favorably in performance with those comeccially available.

The conclusion from this research is that it is possile to use much of the the software developed in the traditional function approach implement an object layer on top of it without a significant loss in performance.

1.2 Client/Server Paradigm

Client/server computing [6] consists of local applications (clients) that access data (server) across a local or wide area network. The client/server paradigm is well established and has been

accepted as the preferred method of implementing distributed applications. A client/server implementation effectively permits a functional split in the processing of a task, as described in Figure 1.

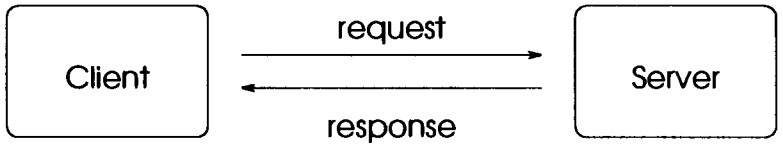


Figure 1: Client/Server Paradigm

1.3 Object Oriented Paradigm

The *object oriented paradigm* is the preferred methodology for systems reengineering because objects provide a tractable way of organising the complexities of an application [18] [4]. Objects, with their natural combination of data and behaviour and strict separation of interface and implementation, make a neat, useful package for distributing data and processes to end-user applications.

The basic object oriented concept is similar to that of the client/server. Each object has associated with it, encapsulated data and methods which implement the functions associated with the object. The methods are invoked by having messages sent to them. The functions operate on the encapsulated data and return the reply as shown in Figure 2.

Thus the object oriented paradigm, in its most simple sense, can be viewed as an implementation of the client/server paradigm. However, in most cases, the object message passed is performed in the same system between objects which reside in the same address space, rather than in different spaces as in the client/server paradigm. Thus, the object oriented paradigm should lend itself to implementation in a client/server distributed manner.

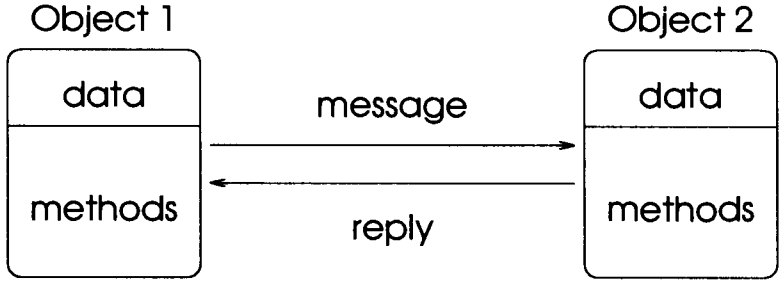


Figure 2: Object Oriented Paradigm

1.4 Distributed Object Technology

This section describes the technologies used to implement client/server computing and the distributed object paradigm. In particular it explains the terms used by the defacto standards

bodies, the Open Software Foundation and the Object Managment Group and sets a terminology basis for the future chapters.

Distributed object computing combines the peer-to-peer client/server paradigm [6] with object oriented technology. An *Object Request Broker (ORB)* is a mechanism for controlling the interaction of distributed objects. That is, the Object Request Brokers communicate with each other by passing messages between individual objects, checking access privileges, providing the security by encryption if required, and prioritising messages. Currently there are a number of implementations of ORBs, although they are mainly proprietary products. Examples of these are IBM's Distributed System Object Model (DSOM) and Iona Technology Ltd's Orbix.

The Object Management Group's (OMG) *Common Object Request Broker Architecture (CORBA)* provides a method of accessing objects regardless of their distributed location.

However, none of these products is implemented using the Open Software Foundations' (OSF) *Distributed Computing Environment (DCE)*. DCE provides a mechanism for programming secure remote procedure calls between clients and servers. It is used in the present work as the basis for communication between the Object Request Brokers.

The CORBA specification, for the development of applications, comprises two chief parts - an *Interface Definition Language (IDL)* and a *Dynamic Invocation Interface (DII)*. Both provide a fundamental service to enable messaging between objects in distributed systems. The *object implementation* is the code and data that actually implements the object. The ORB is responsible for all the mechanisms required to find the object implementation for the request, to prepare the object implementation ro receive the request and to communicate the data making up the request.

The CORBA IDL structures objects so that, when combined with an *Application Programming Interface (API)* for accessing objects at runtime, applications are constructed with prior knowledge of the kinds of objects with which they will internetwork at runtime.

The DII is an API that can be called from C. The DII allows the programmer to dynamically build messages and argument lists at runtime.

The ORBs use DCE to communicate with one another over a network. It was decided to follow the accepted practice and implement a compiler to produce DCE IDL files and C stub code, from the CORBA IDL file, that could be linked in with the application program.

The objectives of the research described in this thesis were :

- To implement a CORBA compliant distributed Object Request Broker for the purpose of investigating the overhead associated with the layering of the functional and object paradigms.
- To design and implement a compiler which maps CORBA object invocations to DCE remote procedure calls.
- To analyse the performance which can be achieved by a DCE based Object Request Broker which is CORBA compliant relative to both commercial and publicly available products not using DCE.

A detailed description of OMG's CORBA is given in Chapter 2 of this thesis, while DCE is discussed in detail in Chapter 3. The design and implementation of my CORBA compliant Object Request Broker and the features it provides are detailed in Chapter 4. Chapter 5 describes in detail the CORBA compiler which provides two mappings : one which utilizes DCE's data marshalling facilities and the other which uses DCE pipes. Chapter 6 discusses all components involved in building dynamic applications. Chapter 7 describes CORBA's object location scheme and how the ORB binds to this object. Chapter 8 contains an example and a performance test which illustrate the use of the CORBA IDL compiler and DII, and the corresponding code produced.

The conclusions drawn from the present research and recommendations for future work are given in the final chapter.

Due to the fact that we were restricted to only one machine running DCE, our ORB has been restricted to where the client and server objects were separate processes on the same machine. Also, the client and server application programs could only be written in C code.

Chapter 2

Common Object Request Broker Architecture (CORBA)

2.1 What is the OMG?

The *Object Management Group (OMG)* was founded in 1989 by 11 companies including Digital, Hewlett Packard, Hyperdesk, NCR and SunSoft. These companies are authors of the *Common Object Request Broker Architecture (CORBA)* specification, Version 1.0, released in October 1991. It was followed in March 1992 by Version 1.1 and in January 1994 by Version 1.2; the group is currently completing revision 2.0 which is due late in 1995. Today, OMG is a consortium of more than 300 hardware, software and end-user companies.

The goal of the OMG is "to adopt interface and protocol specifications that define an object management architecture supporting interoperable applications based on distributed interoperable objects" [9]. In other words, the aim is to adopt a standard for the interoperation of object oriented software across operating systems and platforms in a heterogenous environment.

CORBA is a specification of an *Object Request Broker (ORB)* whose job it is to enable and regulate interoperability between objects and applications. The ORB is part of a larger vision called the *Object Management Architecture (OMA)*.

The OMA specification is OMG's complete vision of the distributed environment. While the CORBA specification focuses solely on the interaction of objects and the mechanisms which enable it, the OMA defines a broad architecture of services and relationships within an environment, as well as the object and reference models [8]. As Figure 3 illustrates, OMA is built upon the ORB services defined by CORBA which provide the interaction model for the architecture. The environment is made richer with the addition of Object Services and Common Facilities, both intended to serve as building blocks for assembling frameworks within which distributed solutions are built. In the diagram the circles are functional programs and the rectangles represent services, each of wich can have multiple entry points as illustrated by the semi-circles.

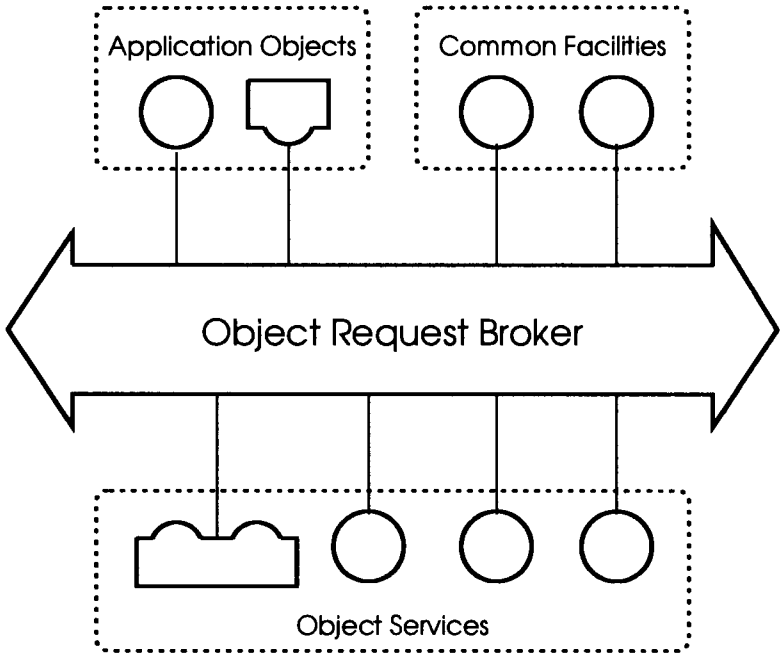


Figure 3: Object Management Architecture Reference Model

Object Services is an area covered by yet another OMG specification, Common Object Services Specification (COSS), that defines a set of objects which perform fundamental operations, such as lifecycle, naming, event, and persistent services. The second stage of the COSS specification defines relationships, externalisation, transactions, and concurrency control. Additional stages planned for the next couple of years will address issues such as security, licensing, queries, and versioning.

Common Facilities (CF) are the newest area of effort by the OMG. Unlike CORBA and Object Services, which are low-level fundamental operations, the Common Facilities has an application-level focus, and defines objects which provide key workgroup support functions : printing, mail, database queries, bulletin boards and newsgroups, and compound documents. The OMG envisages this as the layer most often used by developers working within a distributed environment.

Application Objects are objects specific to particular commercial products or end-user systems.

2.2 Object Request Broker (ORB)

There are two industry stanrad methods of implementing distributed objects. These are OMG's CORBA ORB methodology, and the Microsoft *Common Object Model (COM)* architecture. Of these most manufacturers have adopted the former and produce ORB's wich are CORBA compliant. There are a number of implementations, of commercially available CORBA ORB's some of wich proprietary, but most conform to the OMG's CORBA.

2.3 ORB Architecture Overview

The Object Request Broker architecture can best be described in terms of its components.

Figure 4 [9] below show the various components of the ORB and the communication flow between them.

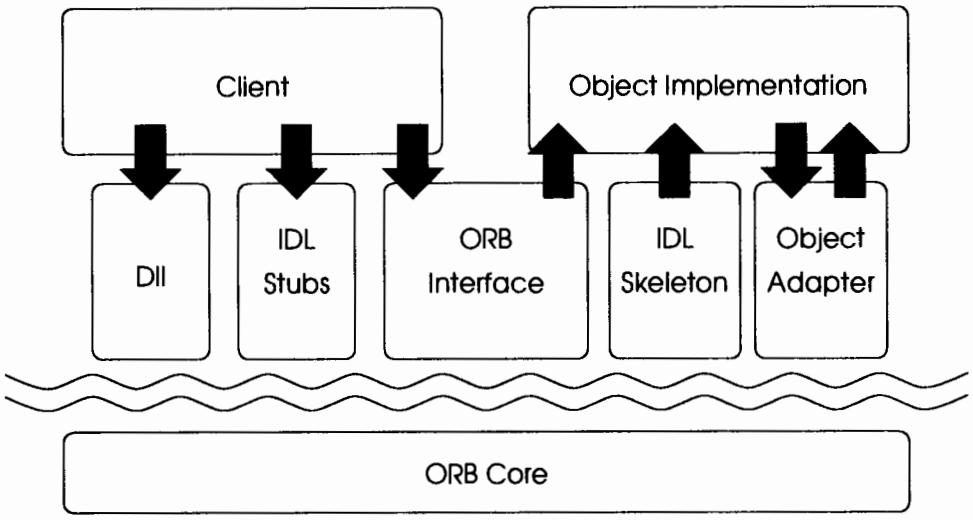


Figure 4: The Structure of an Object Request Broker

2.3.1 Interface Definition Language (IDL)

User application programmers first create IDL interface descriptions for their objects, and then invoke an IDL compiler to generate DCE IDL files, stub and skeleton C code that ends up being executed at runtime during invocations on their objects. Using an object reference, a client may issue requests on the object by making a method invocation using a stub.

A typical IDL file has the following form. The order is unimportant, except that names must be declared (or forward referenced) before they are referenced. These are described in detail in Chapter 5.

Include directives	(optional)
Type declarations	(optional)
Constant declarations	(optional)
Exception declarations	(optional)
Interface declarations	(optional)
Module declarations	(optional)

2.3.2 Dynamic Invocation Interface (DII)

The DII allows applications to invoke methods on an object implementation without having access to the object's stubs and without having an IDL compiler. That is, a client may dynamically compose a request.

2.3.3 The ORB Core

An ORB includes all the communication infrastructure needed to deliver requests and their associated parameters to objects and to return the results to the clients. Although its name suggests a single process or server, an ORB is in fact the embodiment of multiple system entities, including all service location facilities, data marshalling functions, and any other infrastructure that supports the implementation of the CORBA standard.

The communication infrastructure is often referred to as the *ORB core*. The ORB core is responsible for locating objects, handling the connection management between processes and transmitting data associated with requests and responses between client and server processes.

For both the static and dynamic requests, the ORB core locates the object, establishes a connection to the object if necessary and delivers the request and all related information from the client to the object implementation.

2.3.4 The Object Implementation

An object implementation actually provides the services requested by the clients. It is the implementation for the interface defined in IDL. An object implementation may service multiple clients at once. The object implementation can either be started manually through the host operating system's command line interface, or it can be activated automatically by the *Basic Object Adapter (BOA)* when a client wishes to invoke a method on that object.

2.3.5 Basic Object Adapter (BOA)

The *Object Adapter* provides the object implementation access to services provided by the ORB. The BOA is the main interface between the server application and the CORBA runtime environment. In particular, the BOA object handles all communications and interpretation of incoming requests and outgoing responses. When clients send requests to a server, the requests are received and processed by the BOA.

The BOA works together with the server object to create and resolve CORBA references to local objects, and dispatch methods on those objects.

There is one BOA object per server process. Once the target object is instantiated and the BOA has a reference to that object, the BOA delivers any method invocation that is received to the actual object implementation on that object using the method skeleton. The object

implementation then proceeds to fulfil the action requested by the client : it will marshall the results of that method and send a response back to the client.

2.4 The CORBA Object Model

In the CORBA architecture object model, the *clients* of services are isolated from the *providers* of services, or *objects*, through well-defined interfaces. Interfaces to objects are specified using the IDL. The IDL interface definition of an object provides detailed information about the operations permitted on each object that implements that interface, the arguments each operation expects, what it returns and what happens when errors (known as *exceptions*) occur. A client accesses an object by issuing *requests* on the object. From the client's point of view, issuing a request is similar to a method invocation in a conventional C program. A client issues a request by using an *object reference* to the object. The client need not be aware of the location or state of the object.

2.5 Language Mappings

The current CORBA 1.2 specification is specified in terms of a C language binding. This binding is cumbersome and not particularly easy to use. It was hoped to extend my implementation to include the C++ mapping, which will be included in the CORBA 2.0 specification. However, this specification was not yet available. Language Mappings for Smalltalk, Ada, Objective-C and COBOL are also in the process of being standardised by the OMG standards committee.

2.6 Review of ORB Products

2.6.1 ORBeline

ORBeline is a complete implementation of the OMG CORBA specification, using the ONC RPC mechanism developed by Postmodern Computing. It features a complete IDL compiler, an easy to use C++ mapping, full support for threads, an Interface Repository, complete support for the Dynamic Invocation Interface, a Dynamic Directory service, built-in fault tolerance and many other features.

ORBeline is free of charge to Universities for teaching and research purposes.[1]. It currently runs on SunOS 4.x, Solaris 2.3 and HP-UX.

2.6.2 Distributed System Object Model (DSOM)

DSOM from IBM, is an Object Request Broker which conforms to the CORBA standard. DSOM is an object-oriented technology for building, packaging and manipulating binary class libraries.

[1] It can be downloaded from `labrea.stanford.edu`

Class libraries provide the basis for code reuse which is one of the promises of object-oriented programming. This systems runs on multiple platforms but uses the Sun ONC RPC as its basic transport method.

2.6.3 Orbix

Orbix from Iona Technologies have ported their CORBA compliant ORB on top of an existing transaction processing monitor, Tuxedo [2]. It runs in conjunction with Tuxedo and ISIS Reliable Distributed Computing technology, provided by ISIS Distributed Systems, a subsidiary of Stratus Computer. In this way, Orbix provides the object oriented services and the underlying transaction processing facility provides the reliability and resilience required in business critical applications.

2.6.4 Inter-Language Unification system (ILU)

ILU is a multi-language object interface system. The object interfaces provided by ILU hide implementation distinctions between different languages, between different address spaces, and between operating system types. ILU can be used to build multi-lingual object-oriented libraries ("class libraries") with well-specified language-independent interfaces. It can also be used to implement distributed systems and to define and document interfaces between the modules of non-distributed programs. ILU interfaces are specified in ILU's Interface Specification Language, ISL, or in the OMG IDL language specified for CORBA. It is an implementation of CORBA Version 1.1 only.

I spent some time evaluating ILU which is a partial CORBA implementation.[2] ILU currently runs on Unix platforms, and the distributed transport schemes are TCP/IP and UDP/IP. However, I have established that the transport mechanism used is Sun RPC and not OSF's DCE.

The current release of ILU contains support for the programming languages Common Lisp, C++, ANSI C, Modula-3, and Python.

2.6.5 CORBA Compliant Products

Other CORBA compliant products include DEC's Application Control Architecture Services, and Hyperdesk Corporations' Distributed Object Management System but none of the above products uses OSF's DCE as an underlying transport mechanism.

[2]It can be downloaded from `parcftp.parc.xerox.com`.

Chapter 3

Distributed Computing Environment (DCE)

This chapter describes DCE and the relationship between DCE and CORBA, which was discussed in the previous chapter. It also describes the features of DCE that can be used to implement a CORBA compliant ORB.

3.1 What is DCE?

The Open Software Foundation's (OSF) *Distributed Computing Environment (DCE)* [5] [19] [17] is an integrated collection of network services that supports the development, use and maintenance of distributed applications. In other words, DCE provides a high level environment for developing and running applications on a distributed system.

3.2 Overview of DCE Components

Figure 5 shows the different DCE components and how they fit together. DCE resides between the Object Request Broker shown at the top of Figure 5 and the operating system and transport services at the bottom. The boxes outlined with solid lines show components that comprise system administration functions as listed below.

3.2.1 Description of DCE Components

The DCE components fall into two categories :

- *Connection mechinisms for developing distributed applications*
  These comprise the *Remote Procedure Call (RPC)* and DCE Threads.

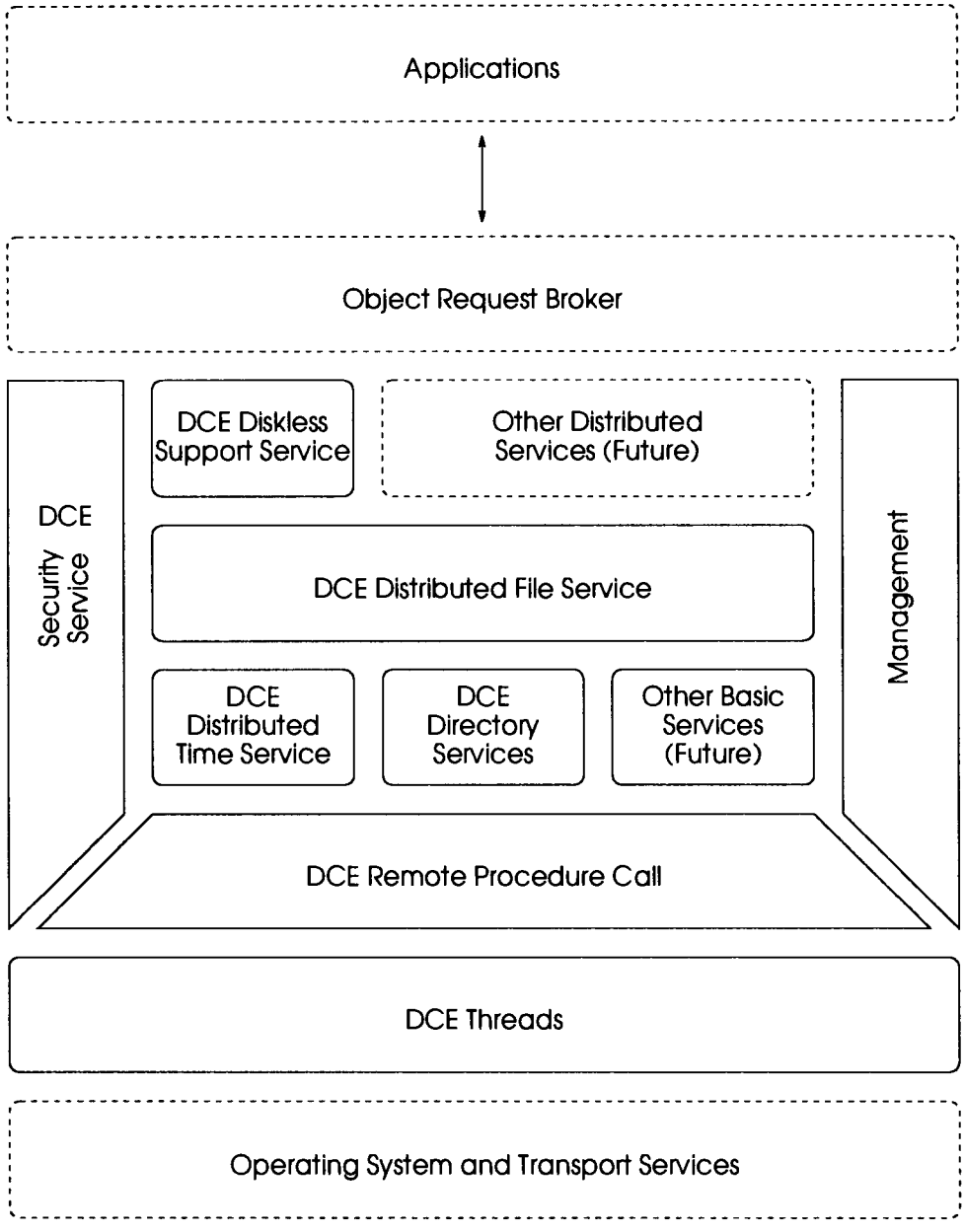


Figure 5: DCE Architecture showing ORB position

- *Services for running distributed applications*
  These services such as the Cell and Global Directory Services (CDS and GDS), the Security Service, Distributed Time Service (DTS),and Distributed File Service (DFS) provide the support required in a distributed system that is similar to that provided in a centralised operating system.

The threads, RPC, CDS, security, and DTS components are commonly referred to as the *"secure core"* and are the required components of any DCE installation. DFS is an optional component which is not available with our version of DCE.

Only the DCE RPC, threads and CDS components are discussed further as the other components have not been used in the implementation of my ORB.

3.2.2 Cell Directory Service (CDS)

In order to help clients find servers in a flexible and portable manner, DCE provides a name service to store binding information. A *name service* is a distributed database service used by applications to store and retrieve information. The CDS is a particular name service supplied with DCE. The RPC runtime library contains name service routines used to indirectly access the DCE Directory Service. RPC servers store binding information in the name service database so that RPC clients can retrieve the binding information and find servers.

3.2.3 DCE Threads

DCE Threads support the creation, management and synchronisation of multiple threads of execution within a single process which is often part of the operating system layer.

3.2.4 DCE Remote Procedure Call (RPC)

A DCE RPC is the primary method of client/server communication. A RPC executes a procedure located in a separate address space from the calling code.

Applications that use RPCs look and behave much like local applications in that all the network communication details are hidden from the application programmer. However, an RPC application is divided into two parts: an RPC server, which offers one or more sets of remote procedures, and an RPC client, which makes remote procedure calls to RPC servers. A server and its clients generally reside on separate systems and communicate over a network.

3.2.5 DCE RPC Interfaces

Traditionally, calling (client) code and called (server) procedures share the same address space and are linked. In an RPC application, the calling code and the called remote procedures are not

linked; rather, they communicate indirectly through an RPC interface. An RPC interface is a logical grouping of operations, data types, and constants that serves as a unique network contract for a set of remote procedures. DCE RPC interfaces are compiled from formal interface definitions written by application developers using the DCE *Interface Definition Language (IDL)*. Each RPC interface contains a Universal Unique Identifier (UUID), which is a hexadecimal number that can identify an entity. A UUID that identifies an RPC interface is known as an interface UUID. The interface UUID ensures that the interface can be uniquely identified across all possible network configurations.

In addition to an interface UUID, each RPC interface contains major and minor version numbers. Together, the interface UUID and version numbers form an interface identifier that identifies an instance of an RPC interface across systems and through time.

The following example shows the use of IDL in a simple interface definition for a banking application

```
[
uuid( 67221A21-EEE6-11CE-8718-0000C0284909 ),
version( 1.0 )
]
interface bank
{
  /* Constant and Data Type Declarations */
  typedef int Boolean;
  typedef [string,ptr] char *string;

  void makeDeposit(
    [in] string name,
    [in] long amount,
    [out] Boolean status );

  void makeWithdrawal(
    [in] string name,
    [in] long amount,
    [out] Boolean status );

  long get_balance(
    [in] string name );
}
```

The above interface definition contains the interface header, constant type declarations, and operation declarations of the `makeDeposit()`, `makeWithdrawal()` and `get_balance()` remote procedures.

3.3 What are the advantages of DCE?

- DCE provides services that can be found in other computer networking environments, but packages them so as to make them much easier to use. For example, the DCE RPC facility provides a way of communicating between software modules running on different systems that is much simpler to code than older methods, such as using socket calls.
- DCE provides new capabilities that go beyond what was available previously. For example, the DCE Security Service provides a reliable way of determining if a user of a distributed system should be allowed to perform a certain action. This is very useful for most distributed applications, yet the design and implementation effort entailed in providing such a capability would be prohibitive for an individual developer.
- DCE integrates components in a manner that makes them more valuable together than separately. For example, the DCE RPC uses threads in such a way that a developer can implement a multi-threaded server without ever explicitly creating or destroying a thread.
- DCE supports both portability and interoperability by providing the developer with capabilities that hide differences among the various hardware, software and networking elements an application will deal with in a large network. For example, the RPC automatically converts data from the format used by one computer to that used by another.

  (Portability is a measure of the ease with which a piece of software that executes on one type of computer can be made to execute on a different type of computer. Interoperability is a measure of the ability of computers of different types to participate in the same distributed system.)

3.4 What is the relationship between DCE and CORBA?

In order to understand the relationship between DCE and the CORBA, it is necessary to understand the evolution from the procedural styles of the past through to the future object paradigms.

Historically, the object paradigm has been viewed as a break with procedural styles of the past. Objects, which encapsulate data and procedures behind an external interface, are often contrasted with other approaches where procedures and data are treated separately. DCE provides a lower-level programming model than CORBA does. DCE is not fully "Object-Oriented".

In this context, DCE is a descendant of the procedural school which emphasises the decomposition of programs into procedures and achieves distribution by locating some of those procedures remotely. Thus there was a tendency for the object community, including the OMG, to view DCE as technology which was obsolete before it was available.

However this view ignored the fact that designers of distributed systems had for a long time recognised that the most successful approach to developing distributed systems was to create

encapsulated objects that can only be accessed via well defined interfaces. Thus the cornerstone of DCE RPC is the interface definition language (IDL) which allows the external attributes of a set of server operations to be specified.

Furthermore, the name-based binding mechanisms of DCE were extended, by the OSF and later versions of DCE to include the ability to bind to a server based on the object instances which it supports. These object binding mechanisms also allow the transparent selection among multiple implementations of the same server operations based on the type of the specified object. In object terminology this is called polymorphism.

The DCE notion of a server supporting interfaces consisting of one or more operations is so close to the notion of an object which provides one or more methods, that it should be no surprise that CORBA 1.2 defines an IDL which differs from DCE IDL in only a few significant respects. In fact, the mapping between these interface languages is a significant part of this thesis as described in Chapter 5.

In CORBA IDL every call must specify an object, which is used in determining the server to use. DCE can do this as well, but there is more work involved and it is optional. Another difference is that CORBA IDL allows an interface to be defined as a extension of one or more other interfaces, this is called interface inheritance.

The use of object oriented techniques and principles should not be confused with using an object oriented language. Object oriented designs can be expressed in procedural languages, and in fact most of the current object environments supported C before supporting C++ or Smalltalk. Therefore, the fact that the DCE *Application Programming Interface (API)* is implemented in C is no barrier to using it to create a distributed object system. In fact, CORBA 1.2 does specify C language bindings.

At present CORBA 1.2 defines only the interface between application components and the object request broker implementation provided by a vendor. Clearly it is intended that objects which interact with the environment provided by a given vendor will be able to interoperate across the systems and networks supported by that vendor. However, CORBA 1.2 does not specify how they will do this. Therefore, there is no basis for interoperation between objects in environments provided by different vendors, nor between heterogeneous brokers.

DCE in contrast, has completely specified the means of interoperation, has in most cases been implemented from the same code base across multiple vendors and has had several public demonstrations of interoperation between many vendor's products.

Further, DCE addresses not only the basic problem of handling requests and responses, but other important distributed systems problems, such as specifying and controlling concurrent execution, providing authentication and access control for security and providing consistent network time services. None of these is yet a part of CORBA 1.2 specification.

Even where DCE lacks features essential to significant numbers of distributed applications, the means of providing them is well understood, and in some cases available today in products which build upon DCE. For example, consider distributed transactions. The means of providing

them via RPC has been known for years and transaction monitors such as Encina and Tuxedo operate over DCE. Consensus has been reached as to how to provide distributed transactions in an object environment. The Object Transaction Service has been developed by various companies within the OMG, which is progressing towards adoption by the *Object Management Architecture (OMA)*.

At the present time, CORBA-compliant products provide for the development of distributed applications only in homogeneous environments and lack many of the capabilities essential to industrial-strength applications. In contrast, DCE provides proven heterogeneous interoperability and most of the capabilities required by robust, production applications.

Most authorities agree that in the long term object technology will be the basis for building large-scale distributed systems [16, 3]. In addition to the principle of encapsulation, object-based systems allow systems to be built up, evolve and be reconfigured as needed because of their ability to dynamically bind requesters to objects that provide services. This is recognised by the OSF as well. Not only has OSF based the Distributed Management Environment (DME) model on object principles and committed to use the CORBA as the underlying implementation tool, but also they have sought to add more object oriented features to DCE and research the use of DCE technologies as a basis for CORBA interoperability. After some years of remaining at arm's length, each of OSF and OMG are now members of the other. Individuals and companies associated with both are taking active roles in working groups of both.

The OMG is now in the process of adding some of the missing pieces of CORBA 1.2 by defining CORBA 2.0. These include the IDL C++ Mapping, the Interface Repository and the Universal Networked Objects (UNO) document [15].

OMG has issued requests for technology in several other areas as well. Some, such as security and time services, are addressed by DCE whereas others, such as lifecycle (creation and deletion) and persistence (database), are not. Ultimately, it is to be hoped that there will be a number of CORBA-compliant products to choose from which interoperate, provide application portability, and offer all the features of DCE and more, as well as the benefits of object-based systems.

DCE is used as the base communications mechanism in my CORBA based Object Request Broker, described in Chapter 4.

3.4.1 Migration between DCE and CORBA

It is possible to employ object techniques when developing distributed applications using DCE. Carefully designed systems will be able to take advantage of DCE features such as dynamic binding and polymorphism and converge with CORBA-compliant systems as they mature.

The likelihood that DCE will be a common base technology for CORBA interoperability, implies that the eventually migration of applications which use DCE directly to an object environment should not present any insurmountable difficulties.

3.5 Interoperability

The standard interoperability protocol for networked ORBs is called the *General Inter-ORB Protocol (GIOP)*. A specific mapping of the GIOP which runs over TCP/IP connections is called the *Internet Inter-ORB-Protocol (IIOP)*. The Internet Inter-ORB-Protocol (IIOP) also "defines the minimum additional protocol layers necessary to transfer CORBA requests between ORBs" [14]. There is also an Environment Specific Inter-ORB Protocol (ESIOP) based on DCE called the *DCE Common Inter-ORB Protocol (DCE-CIOP)*. Figure 6 [15] depicts the relationships between these protocols. The CORBA IDL provides structures which will permit applications to serve as a bridge between the two ORB protocols. This will include data marshalling and converting to a standard data representation. The most common standard representation are the ONC's RPC External Data Representation (XDR) and the DCE Common Data Representation (CDR).

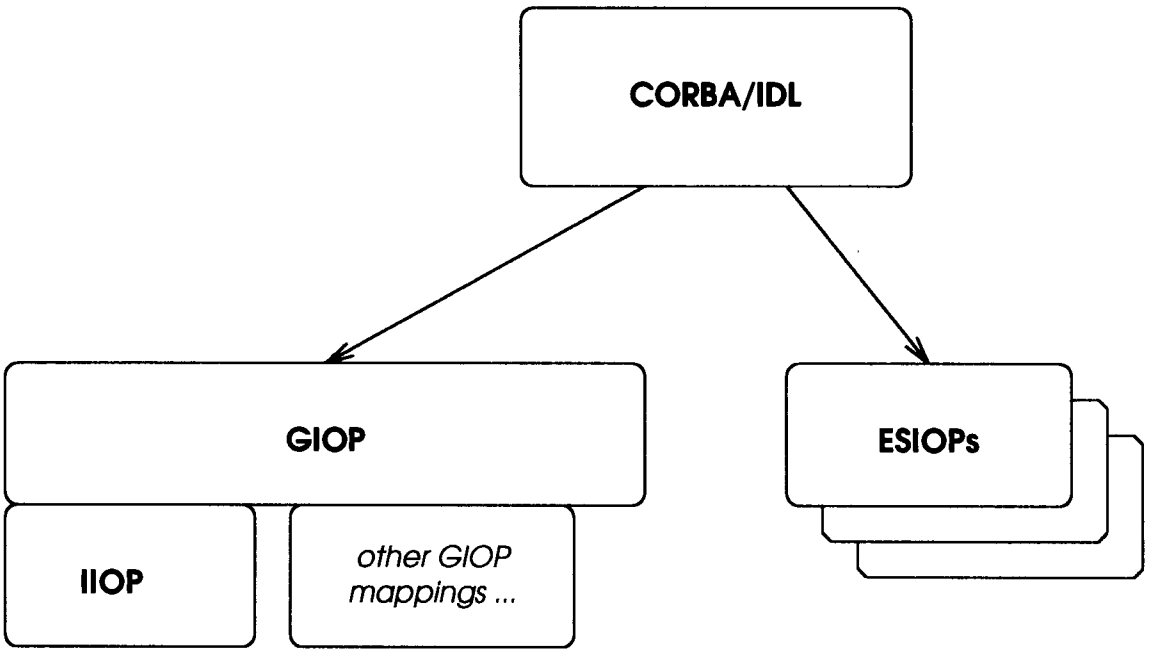


Figure 6: Inter-ORB Protocol relationships

DCE-CIOP message headers and bodies are specified as DCE IDL types. These are encoded using *Common Data Representation (CDR)* and the resulting messages are passed between client and server process via DCE RPC pipes.

SunSoft's "Inter-ORB Engine" is essentially a small portable ORB core.[1] The ORB uses SunSoft's CDR marshalling engine to encode simple DCE IDL data types directly and uses a TypeCode interpreter to marshall complex data types.

For DCE-CIOP to be used to invoke operations on an object, the information necessary to reference an object via DCE-CIOP must be included in an *Interoperable Object Reference (IOR)* which is and IDL structure. DCE-CIOP information is stored in an IOR as a set of components in

[1] It can be downloaded from `ftp://ftp.omg.org/pub/interop/iiop.tar.gz`).

a profile. This is a structure which identifies the ORB. The IOR is created by the server ORB to provide the information necessary to reference the `CORBA_Object`.

Two DCE-RPC interfaces are defined in [15] for the transport of messages between client ORBs and server ORBs. One interface uses pipes to convey the messages, while the other uses conformant arrays. The pipe-based interface is the preferred interface, since it allows messages to be transmitted without precomputing the message length. Only the pipe-based interface has been implemented as "a future DCE-CIOP revision may eliminate the array-based interface" [15].

3.5.1 Pipe-based Interface

The `dce_ciop_pipe` interface is defined by the standard DCE IDL specification shown below :

```
[
uuid( 8CDD5E81-11E3-11CF-8C31-0000C0284909 ),
version( 1.0 )
]
interface dce_ciop_pipe
{
  typedef pipe byte messageType;

  void invoke(
    [in] handle_t bindingHandle,
    [in] messageType requestMessage,
    [out] messageType *responseMessage
  );

  void locate(
    [in] handle_t bindingHandle,
    [in] messageType requestMessage,
    [out] messageType *responseMessage
  );
};
```

The `dce_ciop_pipe` interface is made up of two DCE-RPC operations `invoke` and `locate`. The first parameter of each of these RPCs is a DCE binding handle, which identifies the server process on which to perform the RPC. The remaining parameters of the `dce_ciop_pipe` RPCs are pipes of uninterpreted bytes. These pipes are used to convey messages encoded using CDR. The `requestMessage` input parameters send a request message from the client to the server, while the `responseMessage` output parameters return a response message from the server to the client.

Figure 7 [15] illustrates the layering of DCE-CIOP messages on the DCE RPC protocol using *Network Data Representation (NDR)* pipes.

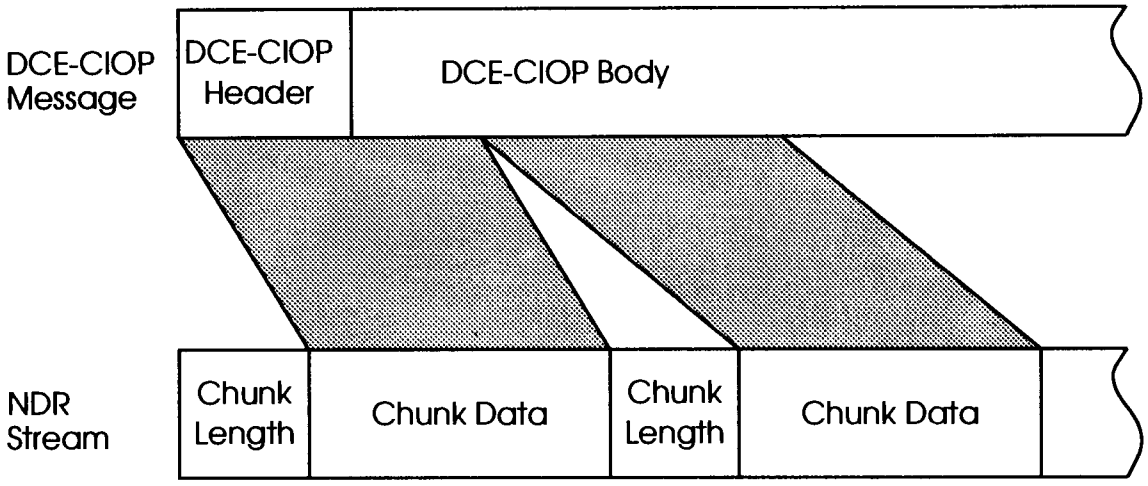


Figure 7: DCE-CIOP Pipe-based Interface Protocol

A pipe is made up of chunks, where each chunk includes a chunk length and check data. The chunk length is an unsigned long indicating the number of pipe elements making up the chunk data. The pipe elements are DCE IDL bytes, which are uninterpreted by NDR. A pipe is terminated by a chunk of length zero. The pipe chunks are concatenated to form a DCE-CIOP message.

The DCE-CIOP message formats are specified in DCE IDL, are encoded using CDR, and are transmitted over DCE RPC using pipes.

Invoke

The `invoke` RPC is used by a client process to attempt to invoke a request in the server process identified by the `bindingHandle` parameter. The `requestMessage` pipe transmits a invoke request message encoded using CDR from the client to the server. The `responseMessage` pipe transmits a invoke response message, also encoded using CDR, from the server to the client. A DCE-CIOP `invoke` request/response message is made up of a header and a body. The header has a fixed format, while the format of the body is determined by the operation's CORBA IDL definition. The format of the invoke response message body also indicates the outcome of the message. The translation of the body from the operation's CORBA IDL format to CDR uses SunSoft's data marshalling routines.

Locate

The `locate` RPC is used by a client process to query the server process identified by the `bindingHandle` parameter for the location of the server process where requests should be sent. The `requestMessage` and `responseMessage` parameters are used similarly to the parameters of the `invoke` RPC. A DCE-CIOP `locate` request message is made up of a fixed-format header and no body. The format locate response message body depends on information in the header.

Chapter 4

The Research Problem - Implementing CORBA within DCE

The research presented in this thesis is a complete implementation of a CORBA compliant ORB with DCE responsible for all the mechanisms required to find the object implementation for the request, to prepare the object implementation to receive the request, and to communicate the data making up the request. This chapter gives a detailed discussion of the components of my Object Request Broker.

Figure 8 shows the basic structure of my DCE-based CORBA compliant ORB. The client performs a request by having access to an object reference for an object and knowing the type of object and the desired operation. The client initiates the request by calling stub routines that are specific to the object or by constructing the request dynamically. Interface information is made available to the client through the Interface Repository and IDL stubs.

The ORB locates the appropriate implementation code, transmits parameters and transfers control to the object implementation through an IDL skeleton. Skeletons are specific to the interface. When the request is complete, control and output values are returned to the client. Implementation information is made available to the object implementation through the Implementation Repository and IDL skeletons. Hence, a CORBA IDL compiler is needed to generate the client stubs and the object implementation specification. The compiler is also used to permeate the Interface Repository with runtime interface definitions.

Object implementation information is stored in the Implementation Repository for use during request delivery.

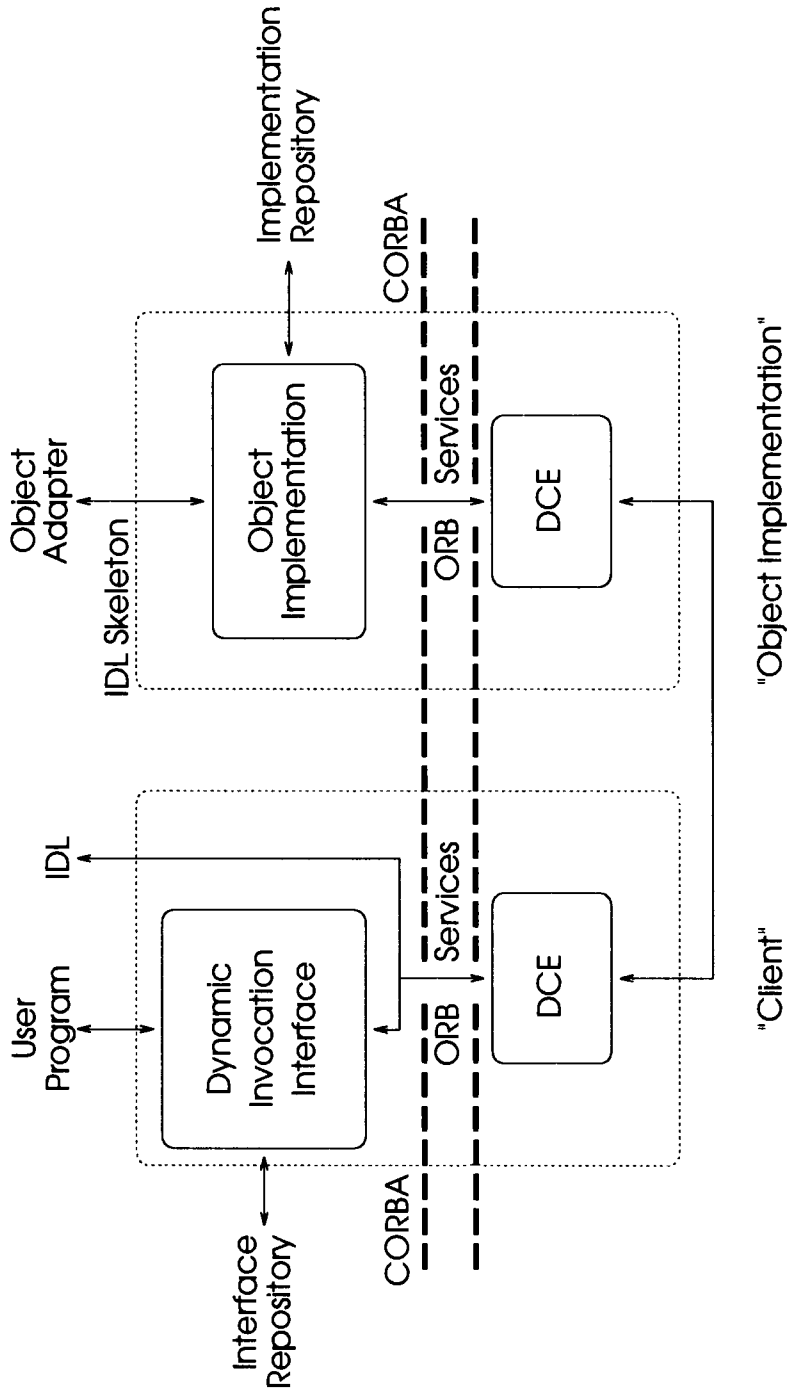


Figure 8: Implementation of DCE Based Object Request Broker

4.1 Structure of the Compiler

The CORBA IDL compiler is implemented using C++, Flex[1] [7] and Bison[2]. The components found in this object-oriented compiler are essentially the same as those found in a traditionally constructed compiler [1].

The *Lexical Analyser (scanner)*, which is written in Flex, basically translates the CORBA IDL source code into a stream of tokens[3] describing the original token lexemes[4]. The *Parser*, which is written in Bison, checks the tokens for correct syntax and then builds an *Abstract Syntax Tree (AST)* along with the appropriate *symbol tables*. The objects out of which this AST is constructed have the ability to interpret the original source program and to produce DCE IDL files and C source code corresponding to the source file.

Figure 9 shows the basic structure of the CORBA IDL compiler. The compiler components include :

- Lexical Analyser.
- Parser.
- Symbol Table.
- Abstract Syntax Tree (AST).
- Semantic Checker.
- Code Generator.

The Lexical Analyser : `yylex()`

The `Flex` utility produces a lexical analyser from the information found in a scanner specification file (See Table 1). This file consists of a header, a set of lexeme definitions, a set of scanning rules, and a collection of user-supplied functions.

Header section
Lexeme regular expression
%%
Scanner rules
%%
User-defined supporting functions

Table 1: Scanner Specification File

[1] Fast Lexical Analyser (Flex) is a freeware tool for generating scanners which can be downloaded from `ftp.sun.ac.za`

[2] Bison is a general purpose parser generator that converts a grammar description for an LALR(1) context-free grammar into a C++ program to parse that grammar. It can be downloaded from `ftp.sun.ac.za`

[3] Tokens are integers that represent categories of various lexemes

[4] Lexemes are 'chunks' of CORBA IDL code

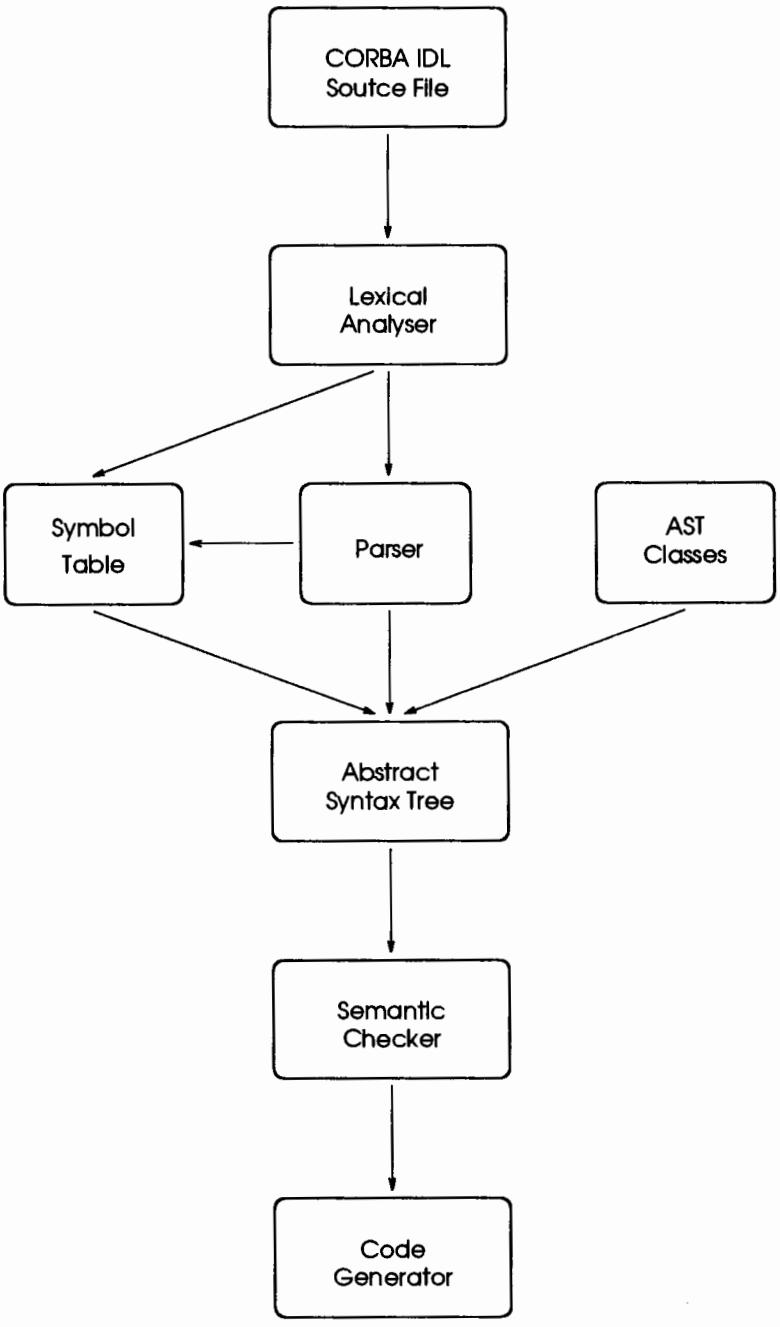


Figure 9: Structure of the CORBA IDL Compiler

Header

Lexemes that the scanner recognises are specified using a very natural kind of symbolism called *regular expressions*. For example, a digit is represented by `{0,1,2,3,4,5,6,7,8,9}` or `[0-9]`. Then, a valid IDL integer could be represented by a set of digits. A valid IDL integer is defined by `[0-9]+` where the + represents one or more of the preceding item; in this case as one or more digits.

Scanner rules

This section of the scanner specification file indicates the actions that the scanner should perform when it recognises a lexeme matching one of the definitions above. After the regular expression is matched, tokens are returned to the parser.

User-supplied functions

Specialised behaviour is often necessary when a particular lexeme is recognised. This last section of the scanner specification file is a convenient location for the declaration of such functions.

The scanner function `yylex()` is produced by Flex from the scanner input file.

The Parser : `yyparse()`

The Bison utility produces a parser from the formation found in the parser specification file (See Table 2).

Header section (optional)
Token specification
%%
Grammar productions
%%
User-defined supporting functions

Table 2: Parser Specification File

Productions

Grammar productions are specified in Backus Naur Form, which describes the composition of the terminal symbols that make up the language rules for parsing CORBA IDL. See Appendix A for the productions which define the entire CORBA IDL.

The syntax checker `yyparse()` is produced by Bison from the parser input file.

The Symbol Table

The compiler uses the symbol table when it needs additional information about any identifier; the compiler performs a `lookup()` of the identifier and finds the information in the form of a

SymbolTableEntry object. The Symbol Table is an object that contains identifiers arranged according to order of encounter along with information about the identifier's purpose, location in the source program, values, etc.

```
typedef enum IdentifierType {
  AST_module,AST_interface, ...
} IdentifierType;
typedef class SymbolTableEntry *SymbolTableEntry_Ptr;
class SymbolTableEntry
{
public:
  SymbolTableEntry();
  SymbolTableEntry( char *_name,IdentifierType );
  int insert( SymbolTableEntry_Ptr );
  SymbolTableEntry_Ptr lookup();

private:
  char *_name;
  IdentifierType _type;
  Scope *_currentScope;
};

typedef class SymbolTable *SymbolTable_Ptr;
class SymbolTable
{
public:
  SymbolTableEntry();
  int insert( SymbolTableEntry_Ptr );
  SymbolTableEntry_Ptr lookup();

private:
  int _tableSize;
  SymbolTableEntry_Ptr *_symTab;
};
```

The symbol table is a list of SymbolTableEntry objects. Since all symbol table entries must be associated with some identifier, the *SymbolTableEntry* constructor defined below requires a string-valued name argument and the identifier type information, which is then stored in the corresponding class member name.

Abstract Syntax Tree

The class definition strategy used is to construct a base class and then define the various leaves of the *Abstract Syntax Tree (AST)*.

AST_BaseClass_Ptr : The Base Class

`AST_BaseClass_Ptr` is the *base class* for all AST nodes. In this case, we say that these nodes are derived from `AST_BaseClass_Ptr` and that they inherit the functionalities of `AST_BaseClass_Ptr`.

```
typedef class AST_BaseClass *AST_BaseClass_Ptr.
class AST_BaseClass
{
public:
  AST_BaseClass();
  virtual int emit();
  virtual LexicalToken *get_LexicalToken() { return _token; };
  // ...

protected:
  LexicalToken *_token;
};
```

The only data member in the definition of `AST_BaseClass_Ptr` is the pointer `_token` to the current `LexicalToken` object. Due the object-oriented design of the AST, the code generator is built into the tree. Instead of every AST node having a separately named `emit()` function as would be required in a procedurally oriented language, we will call all the functions by the same name. Also, because the definition of `emit()` for the base class has been declared virtual for the `AST_BaseClass` base class, the selection of the specific version of `emit()` will be determined at runtime, based upon the nature of the object receiving the `emit()` message! It is not necessary for `AST_BaseClass` to know what kind of definitions are in that linked list because on the basis of the identity of the various objects in that list, the correct `emit()` function is selected and performed at runtime.

Details of the code generator functions will be found in Chapter 5.

It would be tedious to examine each of the leaf classes used in the definition of the (AST). Only a few classes are shown to below as examples.

```
specification :
    definitionLst
    { AST_Specification spec = new AST_Specification( $1 ); }
  ;
```

```
definitionLst :
    definition
    { $$ = new AST_definitionLst( $1 ); }
  | definitionLst definition
    { $$ = AST_definitionLst( $1 )->append( $2 ); }
  ;

definition :
    typeDcl SEMICOLON
    { $$ = new AST_typeDcl( $1 ); }
  | constDcl SEMICOLON
    { $$ = new AST_constDcl( $1 ); }
  | exceptDcl SEMICOLON
    { $$ = new AST_exceptDcl( $1 ); }
  | interface SEMICOLON
    { $$ = new AST_interface( $1 ); }
  | module SEMICOLON
    { $$ = new AST_module( $1 ); }
  ;
```

AST_definition and AST_typeDcl

```
typedef class AST_definition *AST_definition_Ptr;
class AST_definition : public AST_BaseClass
{
public:
  AST_definition();
  virtual int emit();
};

typedef class AST_typeDcl *AST_typeDcl_Ptr;
class AST_typeDcl
{
public:
  AST_typeDcl();
  int emit();
};
```

4.1.1 A Simple Banking Example

Given an IDL interface file as input, the CORBA IDL compiler generates both client stubs and server side skeletons for each method of the interface. In addition, the IDL compiler generates methods used for marshalling the parameters associated with each method.

Consider a simple banking application where the server manages the bank account objects. The following IDL source is defined :

```
/* bankService.idl */

interface account {
  readonly float attribute balance;
  void makeDeposit( in float amount );
  void makeWithdrawal( in float amount );
};
```

The `account` interface has three components: an attribute that represents the current balance in the account, which is *readonly*, and two operations to alter the balance. For simplicity, money is represented as a float, although this would not be a suitable choice in a commercial application due to potential numeric rounding errors.

The attribute `balance` is represented in C as a member function that returns the value of the balance. If the attribute had not been *readonly*, then there would have been a second member function, taking a float argument and returning a void, to set the `balance`.

Note that the `readonly attribute float balance` is logically equivalent to the declaration `float _get_balance()`.

The `bankService.idl` file is passed through an IDL compiler which produces the internal symbol table and abstract syntax tree (shown in Figure 10) which together generate a DCE IDL communication file and/or number of C source files.

4.2 Interoperability

The ORB presented in this thesis can interoperate with any ORB using DCE Common Inter-ORB Protocol, which is discussed in section 3.5. Once an object reference is returned to a client, the client can interact directly with the object in the foreign ORB. This provides true, seamless interoperability.

4.3 Basic Object Adapter (BOA)

My ORB keeps track of all active objects by registering object implementations with the Basic Object Adapter (BOA) daemon. If a client wants to contact that object, the BOA daemon will activate the object implementation and make it available to service client requests.

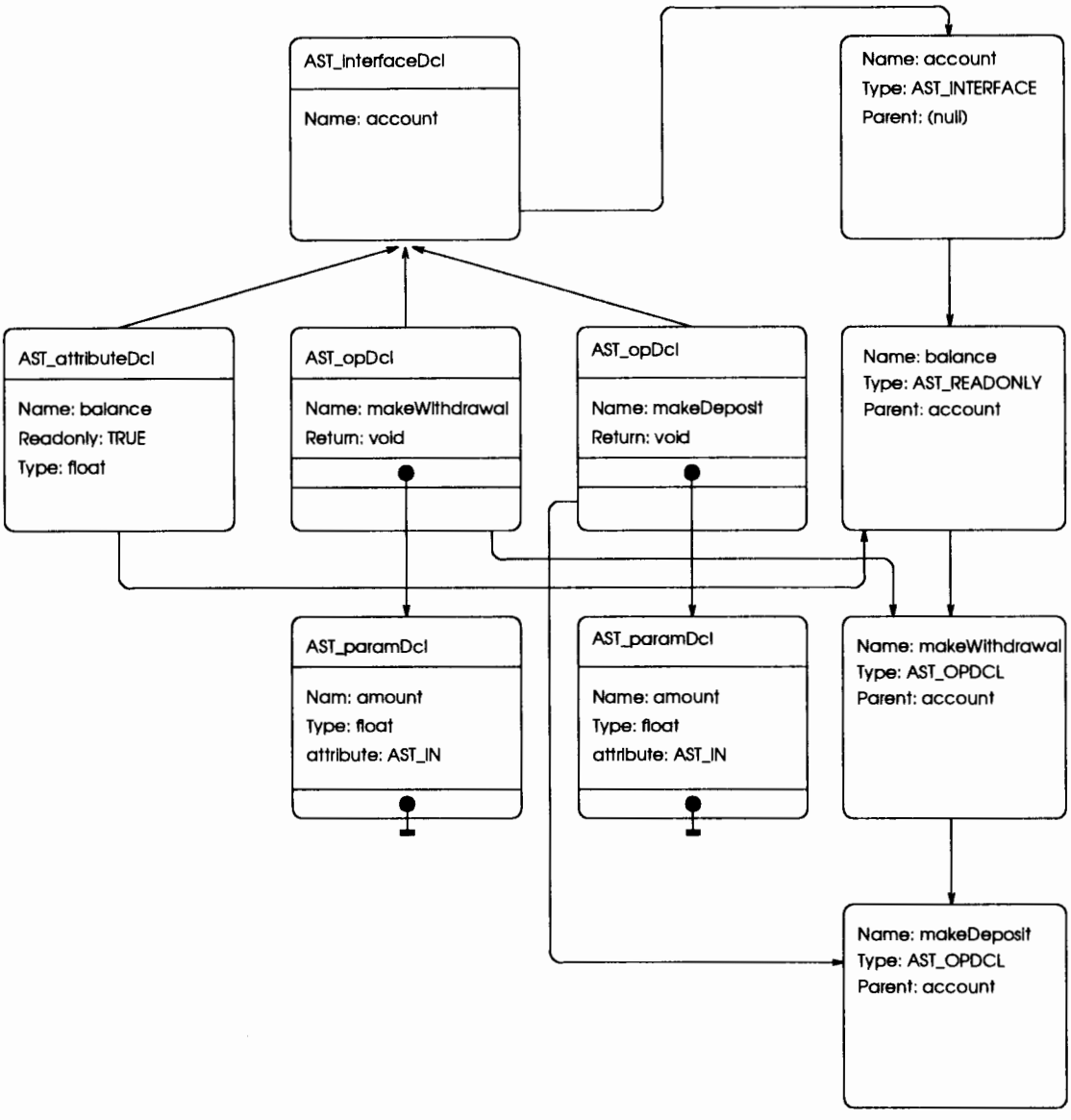


Figure 10: Internal Symbol Table and Abstract Syntax Tree Produced for Simple Banking Example

4.4 Interface and Implementation Repository

My ORB provides a dynamic *Interface Repository*. Information related to CORBA IDL interfaces can be stored in the repository and the repositories can be queried to obtain all relevant information regarding the interface. The interface repository and the dynamic invocation interface can be used to create truly dynamic applications. This is discussed in detail in section 6.3.

My ORB also provides a complete *Implementation Repository* which is managed by the BOA daemon. The repositories can be queried to obtain information related to the implementations. The *Implementation Repository* is discussed in detail in section 6.5.

4.5 Summary

This chapter has given a general description of the components of the DCE based Object Request Broker. In the next chapter, we will discuss the CORBA IDL compiler, the main function of which is to process CORBA IDL files and generate DCE IDL bindings and implementation skeletons.

Chapter 5

CORBA Interface Definition Language (IDL) to DCE IDL Language Mapping

5.1 Introduction

Some of the information in this chapter has already been discussed in one form or another in the previous chapters of this thesis. However, it is important to summarize all information related to the CORBA IDL compiler in one place.

Interface Definition Language (IDL) is the language used to describe the interfaces that clients of objects invoke and object implementations provide. The CORBA IDL language fully describes the operations provided by an implementation. The interface specified in CORBA IDL needs to be translated into DCE IDL files, C stub and skeleton code in order to be used by clients and object implementations.

This chapter describes the DCE IDL and C code generated for the client and object implementation as a result of the compilation of CORBA IDL. Then the two CORBA IDL to DCE RPC mappings for the entire CORBA IDL are represented in detail. The first mapping uses DCE IDL's data marshalling/unmarshalling and the other mapping complies with DCE RPC pipe-based interface.

[The full grammar for CORBA IDL is described in Appendix A.]

5.2 Mapping for DCE IDL Data Marshalling

This section describes in detail the mapping using DCE's data marshalling/unmarshalling facilities. In other words, this section explains how DCE transmits the information involved in

the RPC calls.

5.2.1 Structure of DCE IDL and C Code Generated

For each CORBA IDL interface compiled with the CORBA IDL compiler, a DCE IDL file containing all the data types, constants, typedefs and exceptions declarations and methods for the IDL interface, and four C files are generated. A header and C stub file containing binding code are generated for the client side. Similarly, a header file and C skeleton file are generated for the object implementation.

Figure 11 illustrates the structure of the files generated by the CORBA IDL compiler.

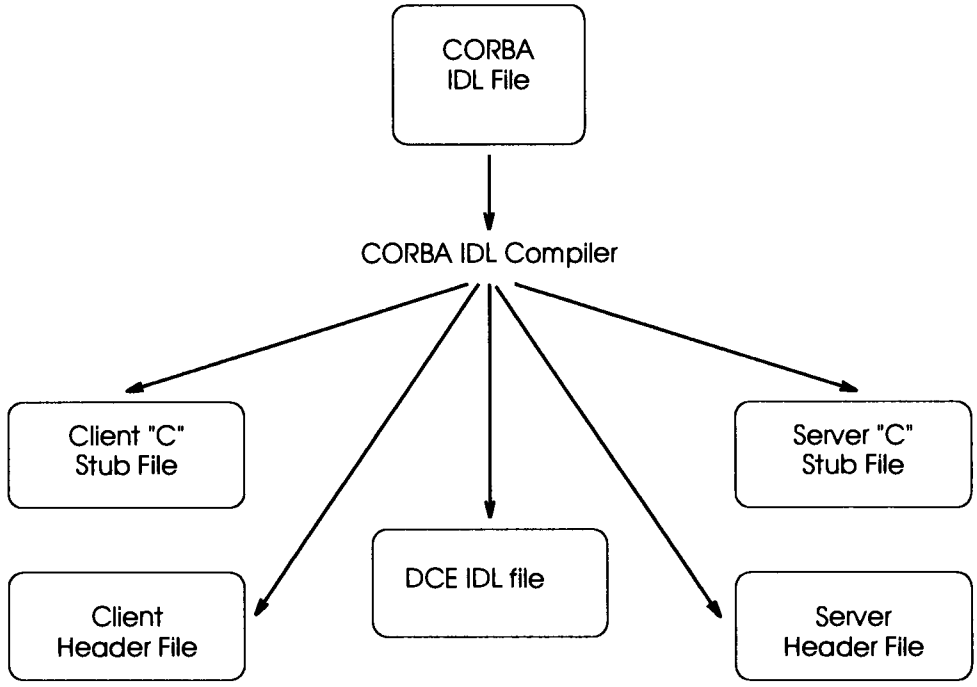


Figure 11: Structure of the Files Generated by the CORBA IDLcompiler using DCE IDL Data Marshalling

5.2.2 Include Directives

The IDL specification for a class normally contains `#include` statements that tell the CORBA IDL compiler where to find the interface definitions (the .idl files) for each of the class's parent classes. For example, if class "C" (defined in file "dove.idl") has a parent "bird" (defined in file "bird.idl") then file "dove.idl" must begin with the following `#include` statement:

```
#include "AB.idl"
```

5.2.3 Mapping for Primitive Data Types

CORBA IDL primitive data types are converted to types defined by DCE IDL. Table 5.2.3) shows the mapping between CORBA IDL primitive data types and DCE IDL types. The DCE IDL code generated by the CORBA IDL compiler is listed in the middle column. The rightmost column shows the equivalent DCE definition corresponding to the C data types which is provided here for the information of the reader.

CORBA IDL	Converted to DCE IDL	Defined as
short	CORBA_Short	short
long	CORBA_Long	long
unsigned short	CORBA_UShort	unsigned short
unsigned long	CORBA_ULong	unsigned long
float	CORBA_Float	float
double	CORBA_Double	double
char	CORBA_Char	char
unsigned char	CORBA_UChar	unsigned char
boolean [a]	CORBA_Boolean	unsigned char
octet	CORBA_Octet	unsigned char
string	CORBA_String	char *
long long	CORBA_LongLong	hyper
long double [b]	CORBA_LongDouble	double
unsigned long long [b]	CORBA_ULongLong	unsigned hyper
wchar [b]	CORBA_WChar	ISO_MULTI_LINGUAL
wstring [b]	CORBA_WString	ISO_MULTI_LINGUAL *

Table 3: Primitive Data Types Mapping

[a] The IDL boolean type is mapped to unsigned char. Two values TRUE and FALSE have been defined and should be the only two values used for this type

[b] See [10] : Appendix A

5.2.4 Mapping for Strings

CORBA IDL strings are mapped to 0-byte terminated character arrays, that is, the length of the string is encoded in the character array itself through the placement of the 0-byte. *Note* that the storage for DCE IDL strings is one byte longer than the stated CORBA IDL bound.

For example, the following CORBA IDL declarations :

```
typedef <string,10> stringExample1;
typedef <string> stringExample2;
```

are defined in the DCE IDL file as a string :

```
typedef CORBA_String *stringExample1;
typedef CORBA_String *stringExample2;
```

5.2.5 Mapping for Attributes

For each attribute defined in a CORBA IDL interface, two methods are defined in the DCE IDL file to provide access to the value of the attribute. One will be used for setting the attribute value and the other for getting the attribute value.

For example, the following CORBA IDL interface

```
/* File : interfaceExample1.idl */
interface interfaceExample1
{
  attribute long attributeExample;
};
```

generates a DCE IDL file called "*interfaceExample1_DCE.idl*"

```
[
uuid( 25979B81-1E0B-11CF-9B95-0000C0284909 ),
version( 1.0 )
]
interface interfaceExample1
{
  /* ... various CORBA definitions ... */

  /* interface interfaceExample1 */
  CORBA_long interfaceExample1_get_attributeExample(
    [in] interfaceExample1 _o,
    [out] CORBA_Environment *_eNv );

  void interfaceExample1_set_attributeExample(
    [in] CORBA_Object _target,
    [in] CORBA_long _attributeExample,
    [out] CORBA_Environment *_eNv );
};
```

For *readonly* attributes only the method for getting the value of the attribute is generated.

5.2.6 Mapping for Structures

CORBA IDL structures are mapped directly onto DCE IDL structures. These are defined in the DCE IDL file.

5.2.7 Mapping for Unions

CORBA IDL discriminated unions are mapped onto DCE IDL structures.

For example, the following union as specified in CORBA IDL :

```
union unionExample switch( long ){
  case '1': octet x;
  case '2': long y;
  default: short z;
};
```

is equivalent to the following DCE IDL structure, which is defined in the DCE IDL file as :

```
typedef struct {
  CORBA_Long _d;
  union {
    CORBA_Octet x;
    CORBA_Long y;
    CORBA_Short z;
  } _u;
} unionExample;
```

The discriminator in the structure is always referred to as _d; the union in the structure is always referred to as _u.

5.2.8 Mapping for Arrays

CORBA IDL arrays map directly to DCE arrays. All array indices run from 0 to size - 1.

For example, the following array as specified in CORBA IDL :

```
typedef short arrayExample[128];
```

is equivalent to the following DCE IDL array which is defined in the DCE IDL file :

```
typedef short arrayExample[128];
```

5.2.9 Mapping for Sequences

A CORBA IDL sequence is a one dimensional array with two characteristics, a maximum size and a length. If the maximum size is not specified, then the sequence is said to be unbounded.

For example, the following CORBA IDL declaration, specified

```
typedef sequence<long,10> sequenceExample;
```

is mapped to the DCE IDL declaration, which is defined in the C data type header file :

```
typedef struct {
  CORBA_ULong _maximum;
  CORBA_ULong _length;
  CORBA_Long *_buffer;
} CORBA_sequence_long;
```

and the following C declaration which is defined in the Client stub header file and the Server skeleton header file :

```
#ifndef _CORBA_sequence_long_defined
#define _CORBA_sequence_long_defined
typedef CORBA_sequence_long CORBA_sequence_long;
#endif /* _CORBA_sequence_long_defined */

typedef CORBA_sequence_long sequenceExample;
```

5.2.10 Mapping for Enums

CORBA IDL enum types map directly to C enums.

For example, the following enum CORBA IDL definition

```
enum enumExample {
  first,
  second,
  third
};
```

has the following DCE IDL mapping which is defined in the DCE IDL file :

```
enum enumExample {
  first,
  second,
  third
};
```

5.2.11 Mapping for Constants

Constant identifiers are mapped directly to a C definition.

For example, the following CORBA IDL declarations

```
const string stringExample = "this is a constant identifier";
const long longExample = 100;
const boolean booleanExample = TRUE;
```

are mapped to the following C definitions respectively :

```
#define stringExample "this is a constant identifier"
#define longExample 100
#define booleanExample TRUE
```

which are declared both in the Client stub header file and the Server skeleton file for compatibility.

5.2.12 Mapping for Typedefs

CORBA IDL `typedefs` are mapped directly to C typedefs.

For example,

```
typedef octet octetExample;
```

is mapped to

```
typedef CORBA_Octet octetExample;
```

and

```
typedef enum enumValues {
  first,
  second,
  third
} enumExample;
```

which is defined in the DCE IDL file :

```
enum enumValues {
  first,
  second,
  third
};
typedef enumValues enumExample;
```

5.2.13 Mapping for TypeCodes

CORBA_TypeCodes are values representing types. CORBA_TypeCodes consist of a *kind* field and a *parameter* list. CORBA_TypeCodes can be obtained from the Interface Repository or can be generated using CORBA's IDL compiler. CORBA_TypeCodes are used in the Dynamic Invocation Interface, they are used by the Interface Repository to represent the type specification of IDL declarations and they are used as part of CORBA_Anys.

```
enum CORBA_TCKind {
  tk_null = 0,tk_void,tk_short,tk_long,tk_ushort,tk_ulong,tk_float,tk_double,
  tk_boolean,tk_char,tk_octet,tk_any,tk_TypeCode,tk_Principal,tk_objref,
  tk_struct,tk_union,tk_enum,tk_string,tk_sequence,tk_array,tk_alias,
  tk_except,tk_longlong,tk_ulonglong,tk_longdouble,tk_wchar,tk_wstring
};

/* interface TypeCode */
typedef struct CORBA_TypeCode {
  CORBA_ULong _length;
  CORBA_Octet *_buffer;
  CORBA_TCKind _kind;
  CORBA_Octet *parameters;
} CORBA_TypeCode;

CORBA_Boolean CORBA_TypeCode_equal(
  CORBA_TypeCode _tCode1,CORBA_TypeCode _tCode2,CORBA_Environment *_eNv );
```

`CORBA_TypeCode_equal()` determines whether two CORBA_TypeCodes are equal.

```
CORBA_TCKind CORBA_TypeCode_kind(
  CORBA_TypeCode _tCode,CORBA_Environment *_eNv )
```

`CORBA_TypeCode_kind()` returns the kind of CORBA_TypeCode.

```
CORBA_Long CORBA_TypeCode_param_count(
  CORBA_TypeCode _tCode,CORBA_Environment *_eNv );
```

`CORBA_TypeCode_param_count()` returns the number of parameters for this CORBA_TypeCode.

```
CORBA_Any *CORBA_TypeCode_parameter(
  CORBA_TypeCode _tCode,CORBA_Long _index,CORBA_Environment *_eNv );
```

`CORBA_TypeCode_parameter()` returns the `_index`'ed parameter. Parameters are indexed from 0 to size - 1.

5.2.14 Mapping for Any

The CORBA IDL CORBA_Any type allows values to be expressed for any CORBA IDL type. A CORBA_Any type maps to the C data type structure CORBA_Any. A CORBA_Any always contains a CORBA_TypeCode. If the CORBA_TypeCode is not `TC_null` or `TC_void`, then the CORBA_Any also contains a value of the type specified by the CORBA_TypeCode. Thus, a CORBA_Any is defined as follows :

```
typedef struct CORBA_Any {
  CORBA_TypeCode _type;
  void *_value;
} CORBA_Any;
```

5.2.15 Mapping for Exceptions

For example, the following is an example declaration of a "BAD FLAG" exception :

```
exception BAD_FLAG {
  long errorCode;
  char reason[80];
};
```

The CORBA IDL compiler maps the above declaration to the following DCE IDL language construct

```
typedef struct BAD_FLAG {
  CORBA_long errorCode;
  CORBA_Char reason[80];
} BAD_FLAG;
```

and the following C definition is declared in the Client stub header file and Server skeleton header file :

```
#define ex_UserExcep_BAD_FLAG "_BAD_FLAG"
```

5.2.16 Mapping for Contexts

A context object contains a list of properties consisting of name-value pairs. Context properties represent information about the client environment or information related to the operation being invoked that cannot easily be sent as a parameter. Context objects can be used when making invocations through the *Dynamic Invocation Interface (DII)* or the C stubs.

5.2.17 Mapping for Interfaces

When CORBA IDL is used to define an interface, a DCE IDL file is generated for invocation of methods. Also, an object UUID is generated for each interface which controls the client and server binding to the ORB.

Again, the code generated is best described through an example. See section 8.1.2 of the "stack" example in Chapter 8.

Interface Inheritance

It is possible for an interface defined in CORBA IDL to inherit from another interface already defined in CORBA IDL. In this case, the code generated for the inheriting class will be a subclass of the parent CORBA IDL interface. The methods, data type definitions, constant and enum declarations that are part of the parent interface will be visible to the sub-interface.

For example, the following CORBA IDL interface :

```
interface interfaceExample1
{
  void op1();
};

interface interfaceExample2 : interfaceExample1
{
  long op2( in short arg1 );
};
```

is mapped to the *"interfaceExample1_DCE.idl"* DCE IDL file

```
[
uuid( BA8F7C81-1E01-11CF-9FE1-0000C0284909 ),
version( 1.0 )
]
interface interfaceExample1
{
  /* ... various CORBA definitions ... */

  /* interface interfaceExample2 */
  typedef CORBA_Object interfaceExample1;
  typedef interfaceExample1 *interfaceExample1_Ptr;

  void interfaceExample1_op1(
    [in] interfaceExample1 _o,
```

```
    [out] CORBA_Environment *_eNv );

};
```

and the "*interfaceExample2_DCE.idl*" DCE IDL file

```
[
uuid( 12DEE881-1E02-11CF-AB6F-0000C0284909 ),
version( 1.0 )
]
interface interfaceExample2
{
  /* ... various CORBA definitions ... */

  /* interface interfaceExample2 */
  typedef CORBA_Object interfaceExample2;
  typedef interfaceExample2 *interfaceExample2_Ptr;

  void interfaceExample2_op1(
    [in] interfaceExample2 _o,
    [out] CORBA_Environment *_eNv );

  void interfaceExample2_op2(
    [in] interfaceExample2 _o,
    [in,out] CORBA_ServerRequest *_request,
    [out] CORBA_Environment *_eNv );

};
```

Therefore, a client can use an object reference of type `interfaceExample2` to invoke `interfaceExample2_op1()` as if `interfaceExample2_op1()` was defined in `interfaceExample2`.

5.2.18 Mapping for Object References

Each CORBA IDL interface is mapped to a DCE IDL interface with the same name. As part of the compilation of the interface, a pointer to that interface is also defined.

For example, in the following CORBA IDL interface :

```
interface objRefExample
{
  void op1();
};
```

the following DCE IDL definitions are generated :

```
[
uuid( 9CE4A161-1E06-11CF-A1BA-0000C0284909 ),
version( 1.0 )
]
interface objRefExample
{

  /* ... various CORBA definitions ... */

  /* interface objRefExample */
  typedef CORBA_Object objRefExample;
  typedef objRefExample *objRefExample_Ptr;

  /* DSI Prototypes ... */

};
```

The typedef `objRefExample` generated is known as the object and the `objRefExample_Ptr` as the object reference. These can be used to perform invocations on the methods provided as part of the interface.

```
CORBA_Environment *_eNv;
objRefExample_Ptr _ex;

/* initialise _ex */
op1( _ex,_eNv );
```

Methods Defined on Object References

A number of methods are always defined on an object reference. For example, for the `objRefExample` interface defined above the following client application stubs are generated :

```
#ifndef _OBJREFEXAMPLE_CSTUB_H_
#define _OBJREFEXAMPLE_CSTUB_H_

#include "CORBA.h"
#include "objRefExample_DCE.h"

objRefExample *objRefExample_bind(
  const char *_objectName,CORBA_Environment * );
```

```
void objRefExample_op1( objRefExample,CORBA_Environment * );

#endif /* _OBJREFEXAMPLE_CSTUB_H_ */
```

The `objRefExample_bind()` method is called by the client application to obtain an object reference of type `objRefExample *` that will be used to perform method invocations. By default when `objRefExample_bind()` is invoked, CORBA will attempt to find an object implementation that matches the interface name of the object reference.

Note that `objRefExample` is a `CORBA_Object`. Therefore it also provides the methods that are defined for all CORBA object references.

- Methods for duplicating and releasing object references :
  - `CORBA_Object *CORBA_Object_duplicate( CORBA_Object, CORBA_Environment * )` duplicates this object reference and returns a pointer to the duplicated object.
  - `void CORBA_Object_release( CORBA_Object *,CORBA_Environment * )` deletes this object.
  - `CORBA_Boolean CORBA_is_nil( CORBA_Object,CORBA_Environment * )` returns TRUE if this is a NIL_OBJECT.
- Method to determine whether a connection has been established between the client and the object implementation : `CORBA_Boolean CORBA_Object_is_bound( CORBA_Object,CORBA_Environment * )`
- Getting object and interface names
  - `CORBA_Char *CORBA_Object_interfaceName( CORBA_Object, CORBA_Environment * )` returns the name of the interface this object supports.
  - `CORBA_Char *CORBA_Object_objectName( CORBA_Object,CORBA_Environment * )` returns the name of this instance of the object. The name is given to the object when it is created.
- Setting and Getting the Principal
  The Principal identifies the client on whose behalf a request is being made. The implementation can use the Principal to filter requests.
  - `CORBA_Principal CORBA_Object_getPrincipal( CORBA_Object,CORBA_Environment )`
  - `void CORBA_Object_getPrincipal( CORBA_Object,CORBA_Principal,CORBA_Environment )`

Data Type	In	Inout	Out	Return
Short	Short	Short *	Short *	Short
Long	Long	Long *	Long *	Long
UShort	UShort	UShort *	UShort *	UShort
ULong	ULong	ULong *	ULong *	ULong
Float	Float	Float *	Float *	Float
Double	Double	Double *	Double *	Double
Boolean	Boolean	Boolean *	Boolean *	Boolean
Char	Char	Char *	Char *	Char
Octet	Octet	Octet *	Octet *	Octet
Enum	Enum	Enum *	Enum *	Enum
objref_Ptr [1]	objref_Ptr	objref_Ptr *	objref_Ptr *	objref_Ptr
Struct	Struct	Struct *	Struct *	Struct
Union	Union	Union *	Union *	Union
String	Char *	Char **	Char **	Char *
Sequence	Sequence *	Sequence **	Sequence **	Sequence *
Array	Array	Array	Array	Array slice *
Any	Any *	Any *	Any **	Any *

Table 4: Arguments and Result Passing

5.2.19 Parameters and Result Calling Conventions

According to CORBA IDL, parameters can be passed with one of the following modes : *in*, *inout* and *out*. Operations on interfaces can also have return values (See Table [9]).

5.2.20 Memory Management Rules

Primitive data types consisting of char, octet, unsigned char, short, unsigned short, long, unsigned long, float, double, boolean, and enums are passed by value as input parameters or as pointers for inout or out parameters. There are no memory management issues revolving around primitive data types. All complex data types (strings, structures, unions, arrays, sequences, object references, and Anys) follow the same memory management rules.

Memory Management for in Parameters

- *Client*
  The client allocates the memory for all the arguments passed as *in* arguments. The ORB on the client side uses it but does not manipulate it.
- *Object Implementation*
  The ORB on the implementation side allocates the memory, the implementation uses it but does not manipulate it. The ORB releases the memory upon returning from the method invocation.

Memory Management for inout Parameters

- *Client*
  The client passes the parameter to the ORB on the client side. The ORB marshalls the parameter and passes it to the implementation. The returned parameter is copied into the memory buffer that the input data occupied. In the case of strings or sequences where the returned parameter might be larger, the ORB will automatically reallocate a larger memory buffer for the returned parameter if necessary. The ORB on the client side will not delete the parameter passed. The client is responsible for deleting the parameter when it is no longer needed.
- *Object Implementation*
  On the server side, the ORB allocates the memory required for the input data. The data is passes to the implementation. The implementation modifies the contents of the parameter passed in and returns it to the ORB. Once the parameter is marshalled by the ORB, the parameter is deleted on the server side of the ORB.

Memory Management for out Parameters

- *Client*
  The ORB on the client side allocates memory for the *out* parameter. The data is copied into that buffer by the ORB when the response to the invocation is received. The client is responsible for deleting that object.
- *Object Implementation*
  On the server side, the implementation code initialises the reference to the pointer to the valid *out* parameter buffer. Once the ORB on the server side has marshalled the data associated with that parameter, the parameter is deleted by the ORB.

Memory Management for return Parameters

- *Client*
  The ORB on the client side allocates the data required for the parameter and passes it to the client. The client is responsible for deleting the parameter.
- *Object Implementation*
  On the server side the implementation allocates the return parameter and passes it to the ORB on the server side. The ORB deletes the parameter after it has been marshalled.

5.2.21 Exception Handling

All system and user exceptions are recorded in the CORBA_Environment structure as a CORBA_Exception. CORBA_Environment is used to report exceptions that occur during the

course of an invocation. The CORBA_Environment interface provides methods for checking whether an exception has occurred and obtaining the value of the exception.

The CORBA_Environment interface is defined as follows :

```
enum CORBA_CompletionStatus {
  COMPLETED_YES,
  COMPLETED_NO,
  COMPLETED_MAYBE
};

enum CORBA_ExceptionType {
  NO_EXCEPTION,
  USER_EXCEPTION,
  SYSTEM_EXCEPTION
};

typedef char *CORBA_ExceptionCode;

typedef struct CORBA_ExceptionBody {
  CORBA_ULong _minor;
  CORBA_CompletionStatus _completed;
} CORBA_ExceptionBody;

typedef struct CORBA_Environment {
  CORBA_ExceptionType _major;
  CORBA_ExceptionCode *_ex;
  CORBA_ExceptionBody *_ptr;
} CORBA_Environment;

CORBA_Char *CORBA_Exception_id( CORBA_Environment_Ptr );
void *CORBA_Exception_value( CORBA_Environment_Ptr );
void CORBA_Exception_free( CORBA_Environment_Ptr );

CORBA_Boolean CORBA_Environment_checkException( CORBA_Environment_Ptr );
void CORBA_Environment_clear( CORBA_Environment_Ptr );
CORBA_Boolean CORBA_Environment_is_nil( CORBA_Environment_Ptr );
```

User Exceptions

User exceptions are defined as part of a CORBA interface. These are mapped to equivalent C definitions which are described in Section 5.3.4.

System Exceptions

In addition to user-defined exceptions, there are several predefined exceptions for system runtime errors. The standard exceptions as prescribed by CORBA are shown in Table 5 [9]. These exceptions correspond to standard runtime errors that may occur during the execution of any method (regardless of the list of exceptions listed in its IDL specification).

Each of the standard exceptions has the same structure: an error code (to designate the subcategory of the exception) and a completion status code. The typedef `CORBA_ExceptionBody` is used to define the standard exceptions.

Exception Name	Description
UNKNOWN	"an unknown exception"
BAD_PARAM	"an invalid parameter was passed"
NO_MEMORY	"dynamic memory allocation failure"
IMP_LIMIT	"implementation limit exceeded"
COMM_FAILURE	"communication failure"
INV_OBJREF	"invalid object reference"
NO_PERMISSION	"no permission for attempted operation"
INTERNAL	"ORB internal error"
MARSHAL	"error marshalling parameter or result"
INITIALIZE	"ORB initialisation failure"
NO_IMPLEMENT	"operation implementation unavailable"
BAD_TYPECODE	"bad typecode"
BAD_OPERATION	"invalid operation"
NO_RESOURCES	"insufficient resources for operation"
NO_RESPONSE	"response to request not yet available"
PERSIST_STORE	"persistent storage failure"
BAD_INV_ORDER	"routine invocations out of order"
TRANSIENT	"transient failure - reissue request"
FREE_MEM	"cannot free memory"
INV_IDENT	"invalid identifier syntax"
INV_FLAG	"invalid flag was specified"
INTF_REPOS	"error accessing interface repository"
BAD_CONTEXT	"error processing context object"
OBJ_ADAPTER	"failure detected by object adapter"
DATA_CONVERSION	"data conversion error"

Table 5: System Exceptions

5.3 Mapping for UNO-specified DCE Pipes

This section is the preferred mapping specified in the CORBA Version 2.0 document which was unavailable at the time that the work in this thesis was implemented, but has subsequenlty been rreleased. Basically, this method uses DCE pipes which transmits large amounts of information efficiently over a network. However, there is more work involved in generating the binding files which are now responsible for the data marshalling/unmarshalling.

5.3.1 Structure of DCE IDL and C Code Generated

For each CORBA IDL interface compiled with the CORBA IDL compiler, a common header file containing all the data types, constants, typedefs and exceptions declarations for the IDL interface, and four C files are generated. A header file and a C stub file containing the stubs and marshalling code are generated for the client side. Similarly, a header file and a C skeleton file containing server skeletons are generated for the object implementation.

Figure 12 illustrates the structure of the files generated by the CORBA IDL compiler.

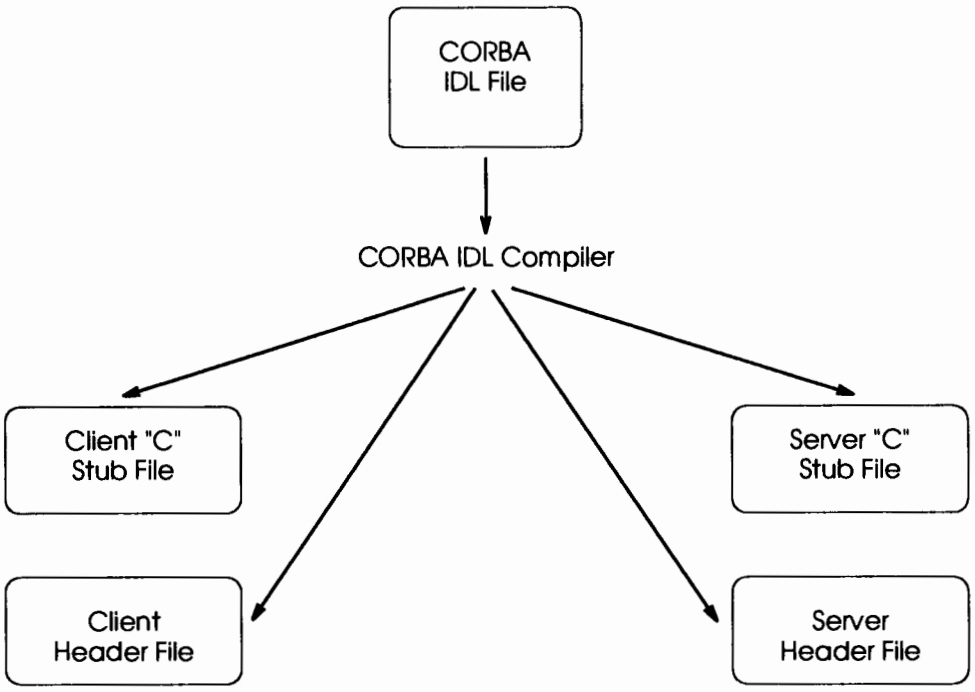


Figure 12: Structure of the Files Generated by the CORBA IDL compiler using UNO-specified DCE Pipes

5.3.2 Mappings for Primitive and Complex Data Types

Mapping for primitive data types, strings, structures, unions, arrays, sequences, enums, constants, typedefs, typeCodes, anys, contexts and object references are the same as for the UNO DCE Pipe mappings.

5.3.3 Mapping for Attributes

For each attribute defined in a CORBA IDL interface, two stubs are generated to provide access to the value of the attribute. One will be used for setting the attribute value and the other for getting the attribute value. Similarly, on the object implementation side two skeletons will be generated for setting and getting the values of the attribute.

For example, the following CORBA IDL interface

```
interface interfaceExample1
{
  attribute long attributeExample;
};
```

Two client stub prototypes are generated for setting and getting the value of the `attributeExample` attribute

```
#ifndef _INTERFACEEXAMPLE1_CSTUB_H_
#define _INTERFACEEXAMPLE1_CSTUB_H_

#include "CORBA.h"
#include "interfaceExample1_IDLTypes.h"
#include "interfaceExample1_DCE.h"

CORBA_Long interfaceExample1_get_attributeExample(
  interfaceExample1,CORBA_Environment * );
void interfaceExample1_set_attributeExample(
  interfaceExample1,CORBA_Long,CORBA_Environment * );

#endif /* _INTERFACEEXAMPLE1_CSTUB_H_ */
```

For *readonly* attributes only the C stubs for getting the value of the attribute are generated. The same holds for the skeletons on the object implementation side.

5.3.4 Mapping for Exceptions

Exceptions are used for reporting errors. Two types of exceptions exist; system and user exceptions. CORBA ORB core can only raise system exceptions. Object Implementations can raise user and system exceptions.

CORBA defines a set of well known system exceptions. User exceptions are defined as part of a CORBA interface.

CORBA IDL specifications may include exception declarations, which define data structures to be returned when an exception occurs during the execution of a method. Associated with each type of exception is a name, and *optionally* a struct-like data structure for holding error information.

For example, the following is an example declaration of a "BAD FLAG" exception :

```
exception BAD_FLAG {
  long errorCode;
  char reason[80];
};
```

The CORBA IDL compiler maps the above declaration to the following C language constructs

```
typedef struct BAD_FLAG {
  CORBA_long errorCode;
  CORBA_Char reason[80];
} BAD_FLAG;
#define ex_UserExcep_BAD_FLAG "_BAD_FLAG"
```

5.3.5 Mapping for Interfaces

When CORBA IDL is used to define an interface, stubs and skeletons are generated for invocation of methods.

Again, the code generated is best described through an example. See section 8.1.2 of the "stack" example in Chapter 8.

Interface Inheritance

For example, the following CORBA IDL interface :

```
interface interfaceExample1
{
  void op1();
};

interface interfaceExample2 : interfaceExample1
{
  long op2( in short arg1 );
};
```

is mapped to the "*interfaceExample1_SSkeleton.h*" server file

```
#ifndef _INTERFACEEXAMPLE1_SSKELETON_H_
#define _INTERFACEEXAMPLE1_SSKELETON_H_
{
/* ... various CORBA definitions ... */

/* interface interfaceExample1 */
void interfaceExample1_op1_SSkeleton(
  CORBA_Object _target,
  CORBA_ServerRequest *_request,
  CORBA_Environment *_eNv );
#endif /* _INTERFACEEXAMPLE1_SSKELETON_H_ */
```

and the "*interfaceExample1_CStub.h*" client file

```
#ifndef _INTERFACEEXAMPLE1_CSTUB_H_
#define _INTERFACEEXAMPLE1_CSTUB_H_

/* ... various CORBA definitions ... */

/* interface interfaceExample1 */
void interfaceExample1_op1(
  interfaceExample1,CORBA_Environment * );
#endif /* _INTERFACEEXAMPLE1_CSTUB_H_ */
```

and the "**interfaceExample2_SSkeleton.h**" server file

```
#ifndef _INTERFACEEXAMPLE2_SSKELETON_H_
#define _INTERFACEEXAMPLE2_SSKELETON_H_
{
/* ... various CORBA definitions ... */

/* interface interfaceExample2 */
void interfaceExample2_op1_SSkeleton(
  CORBA_Object _target,
  CORBA_ServerRequest *_request,
  CORBA_Environment *_eNv );

void interfaceExample2_op2_SSkeleton(
  CORBA_Object _target,
  CORBA_ServerRequest *_request,
  CORBA_Environment *_eNv );
#endif /* _INTERFACEEXAMPLE2_SSKELETON_H_ */
```

and the "`interfaceExample2_CStub.h`" client file has the following function declarations

```
#ifndef _INTERFACEEXAMPLE2_CSTUB_H_
#define _INTERFACEEXAMPLE2_CSTUB_H_

#include "CORBA.h"
#include "interfaceExample1_CStub.h"

void interfaceExample2_op1(
  interfaceExample2,CORBA_Environment * );
CORBA_Long interfaceExample2_op2(
```

```
  interfaceExample2,CORBA_Short,CORBA_Environment * );

#endif /* _INTERFACEEXAMPLE2_CSTUB_H_ */
```

Therefore, a client can use an object reference of type `interfaceExample2` to invoke `interfaceExample2_op1()` as if `interfaceExample2_op1()` was defined in `interfaceExample2`.

5.4 Summary

This chapter illustrates the mapping of the CORBA IDL data representation to those of the DCE IDL. In most cases both syntactical and symantic interpretations are required to carry out this transformation.

Chapter 6

Dynamic Invocation Interface

This chapter discusses all components involved in building dynamic applications. Details are given on the Dynamic Invocation Interface (DII) and how to use it and a section discussing the Interface Repository, the Dynamic Skeleton Interface (DSI) and the Implementation Repository are included.

6.1 Dynamic Invocation Interface (DII)

As we have seen, IDL is used to describe interfaces, and the IDL compiler is used to generate the necessary support from the IDL definitions to allow clients to invoke remote servers. Specifically, the IDL compiler automatically build the appropriate code to manage proxies, to dispatch incoming requests within a server, and to manage the underlying CORBA services.

However, the use of the IDL compiler in this way is a limiting approach for a small but important subset of applications. The IDL interfaces which a client program can use are determined when the client program is compiled. Hence the client code is limited to using those servers which provide the IDL interfaces selected by the client programmer when building an application. For example, a browsing tool with a fixed set of pre-defined interfaces would be undesirable as it would be unable to browse any new structures which were dynamically added.

Therefore CORBA supports a DII which allows an application to issue requests for any interface, even if that interface was unknown at the time the application was compiled.

CORBA allows invocations to be constructed at runtime by specifying at runtime the target reference, the operation name and the parameters. Such calls are termed "dynamic" because the IDL interfaces used by a program do not have to be "statically" determined at the time the program is designed and implemented.

It is important to note that a server receiving an incoming invocation does not know, or care, whether the client which sent the request used the static or dynamic approach to compose the request.

6.2 Using the DII

The application programmer can make dynamic invocations using the functions defined in the `Request` interface which is specified in Section B.1 of Appendix B.

The simple banking application shows how a client can make dynamic invocations. Recall the basic CORBA IDL for the bank example from Section 4.1.1 :

```
interface account {
   readonly float attribute balance;
    void makeDeposit( in float amount );
    void makeWithdrawal( in float amount );
};
```

It is assumed that we already have some server which implements the above interfaces.

The following client program retrieves John Smith's bank balance using the DII :

```
#include "DII.h"
main()
{
  account John_Smith;
  CORBA_Environment *_eNv;

  CORBA_DII_Request *_reQuest;
  CORBA_NVList *_argList;
  CORBA_Identifier _name;
  CORBA_Long _length;
  CORBA_Float _result;
  CORBA_OperationDef _opDef;
  CORBA_OperationDescription *_opDesc;
  CORBA_Contained_Description _desc;
  CORBA_TypeCode _tc;

  CORBA_ORB_init( &_eNv );

  /* ... */

  /* Equivalent IDL request - account_get_balance( John_Smith,&_eNv ); */

  /* Get the OperationDef from the Interface Repository */
  _opDef = CORBA_Repository_lookup_id( CORBA_InterfaceRepository_Obj,
                                       "account_get_balance",
```

```
                                        &_eNv );

  /* Create a NamedValue list for the operation */
  _argList = ( CORBA_NVList * )NULL;

  /* Get the operation description structure */
  _desc = CORBA_OperationDef_describe( _opDef,&_eNv );
  _opDesc = ( CORBA_OperationDescription * )_desc.value._value;

  /* Fill in the TypeCode field for the result */
  _result.argument._type = _opDesc->result;

  /* Create the request, CORBA_DII_Request_Obj */
  _reQuest = CORBA_DII_Request_create_request(
    _stk,"stack_push",_argList,_resultNV,&CORBA_DII_Request_Obj,
    ( CORBA_Context * )NULL,&_eNv );

  /* Finally, invoke the request */
  _reQuest = CORBA_DII_Request_invoke( CORBA_DII_Request_Obj,
                                       ( CORBA_Flags )0,
                                       &_eNv );

  printf( "Balance on John Smith's account is %d\n",
    *( CORBA_Float * )_result.argument._value );

  CORBA_ORB_uninit( &_eNv );
};
```

6.3 The Interface Repository (IR)

This section is based on the OMG Technical (TC) Document 94-11-7 which represents a merger of the two original submissions, one by Digital and HP (OMG TC Document 94-5-3) and the other by SunSoft (OMG TC Document 94-5-2). The Interface Repository (IR) is a database containing all publicly defined interfaces specified in IDL. The IR maintains a collection of interface objects. Interface objects consist of the following [11]:

- **Repository** - the top level module for the repository name space, containing constants, typedefs, exceptions, interface definitions and modules.
- **ModuleDef** - a logical grouping of interfaces; it contains constants, typedefs, exceptions, interface definitions and other modules.

- **InterfaceDef** - an interface definition; it contains lists of constants, types, exceptions, operations and attributes.
- **AttributeDef** - the definition of an attribute of the interface.
- **OperationDef** - the definition of an operation on the interface; it contains a list of parameters and exceptions raised by this operation.
- **ParameterDef** - the definition of an argument to an operation.
- **TypedefDef** - the definition of a named type that is not an interface.
- **ConstantDef** - the definition of a named constant.
- **ExceptionDef** - the definition of an exception that can be raised by an operation.

Figure 13 shows the structure of the interface to the IR. [A complete CORBA IDL definition of the IR can be found in Section B.2 of Appendix B].

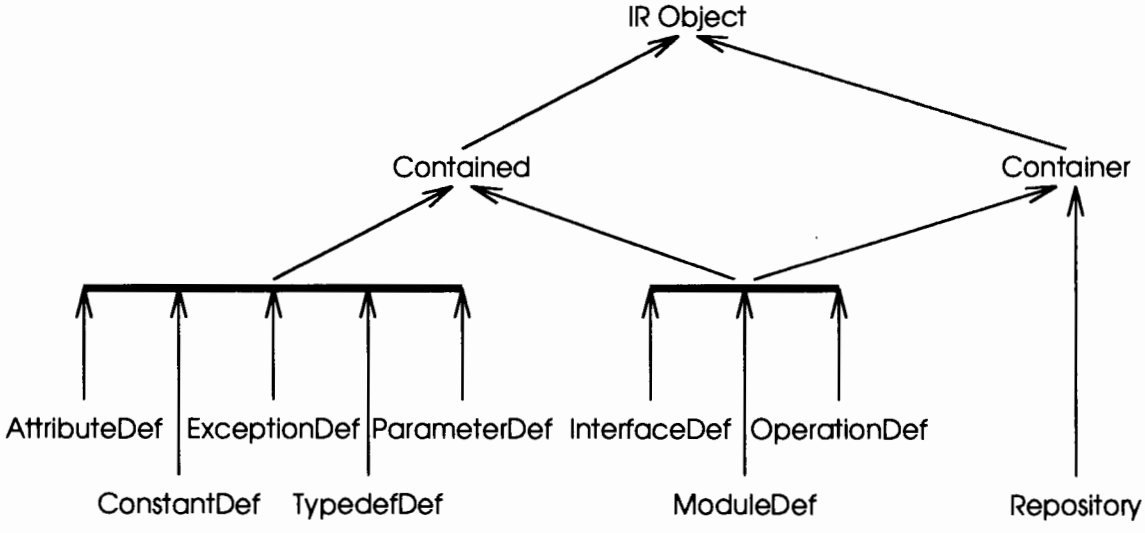


Figure 13: Structure of interfaces to the Interface Repository

6.3.1 Loading Interfaces in the Interface Repository

Interface definitions and descriptions are stored in the IR. The CORBA IDL compiler is responsible for loading entries in the IR. The compiler parses IDL files, generates data structures representing the interfaces and stores them in the IR.

6.3.2 Navigating the Interface Repository

Interfaces in the IR can be queried in two ways. The CORBA_InterfaceDef object that corresponds to a particular repository identifier can be located. Or, the Interface Repository can be navigated using a sequence of names by starting at the root module. Navigating a module can be useful when information about a particular named interface is desired. Starting at the root module of the repository, entries can be obtained by name. Locating the CORBA_InterfaceDef object by the repository identifier is useful when looking for a specific entry in a repository.

6.3.3 Binding to an Interface Repository

To navigate the IR, clients must obtain a handle on the IR object. This is done by calling `CORBA_Repository_bind`. The following provides an example of how to bind to the IR.

```
#include <IR.h>

main()
{
  ...
  CORBA_IR_Repository *_repository = CORBA_IR_Repository_bind();
  ...
}
```

6.3.4 Programming Interface to the Interface Repository

Once bound, the client can navigate the IR or query it. All operations supported on the IR have been specified in CORBA IDL in the CORBA specification. The corresponding code generated using the IDL compiler is included in Appendix B.

All interfaces defined to access the IR are encapsulated in the `CORBA_IR` interface. The `Repository` interface itself inherits from the `Container` interface. The `lookup_id()` method of the `Repository` interface takes a `search_id` as input argument and returns a `Contained` as parameter. The `Contained` interface is also defined within the `CORBA_IR` interface.

To lookup any definition in the IR using an id, the following sequence of code can be used :

```
#include "CORBA.h"

main()
{
  ...
```

```
  CORBA_IR_RepositoryId _searchId;
  CORBA_IR_Contained *_contained = CORBA_lookup_id( _repository,_searchId );

  ...
};
```

Once the `lookup_id()` returns successfully, information such as the name of class or module this interface is defined in can be retrieved using the methods defined on the `Contained` interface. It is possible to invoke the `describe_contents()` for instance to obtain further details on the interface itself.

6.4 Dynamic Skeleton Interface (DSI)

The DSI is a way to deliver requests from an ORB to an object implementation that does not have compile-time knowledge of the type of the object it is implementing. This contrasts with the type-specific OMG IDL-based skeletons in CORBA, but serves the same architectural role.

DSI is the server side's analogue of the client side's DII. Just as the implementation of an object cannot distinguish whether its client is using type-specific stubs of the DII, the client who invokes an object cannot determine whether the implementation is using a type-specific skeleton or the DSI to connect the implementation to the ORB (See Figure 14 [13]).

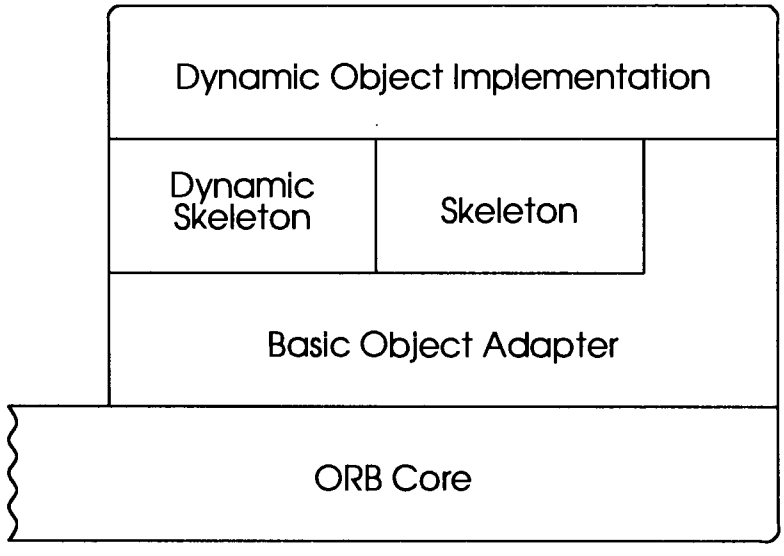


Figure 14: Dynamic and Static Requests are delivered through skeletons

"The DSI implements all requests on a particular object by having the ORB invoke the same upcall routine, a *Dynamic Implementation Routine (DIR).*" [13] A DIR is a function with this definition :

```
typedef void ( *DynamicImplementationRoutine ) (
```

```
  CORBA_Object _target,
  CORBA_ServerRequest _request,
  CORBA_Environment *_eNv );
```

The DIR is passed all the operation parameters, as well as an indication of the object that was invoked and the operation that was requested. This information is encoded in the `_request` parameter. `_target` is the object reference to which the invocation is directed.

`CORBA_ServerRequest` stores the explicit state of a request for the DSI which is analogous to the `CORBA_Request` structure in the DII. It is defined as follows :

```
typedef struct CORBA_ServerRequest {
  CORBA_Identifier _opName;
  CORBA_OperationDef *_opDef;
  CORBA_Context _ctx;
  CORBA_NVList **_params;
  CORBA_NamedValue _result;
  CORBA_Environment *_eNv;
} CORBA_ServerRequest;
```

`_opName` is the name of the operation being invoked. `_opDef` contains the IR entry corresponding to the operation being performed. `_ctx` is the context information for the operation. Before the DIR is called the `_params` is initialised with the TypeCodes for the `in` and `inout` parameters. The DIR will then process the call, producing any result values. If no exceptions are reported, the the DIR will replace pointers to `inout` values in parameters with the values to be returned, and assign pointers to `out` values in `_params`. `_result` stores any return value for the call. `_eNv` is used to report exceptions, both user and system, to the client who made the original call.

During an object invocation, all the typing information is provided to the `CORBA_ServerRequest` by the ORB.

6.5 Implementation Repository

CORBA includes a distributed implementation repository. The `ImplementationDef` of all implementations registered with the BOA daemon is stored in the implementation repository. Before an object implementation (server program) can be used by client applications, it must be registered with CORBA. This is discussed in detail in 7.

6.5.1 ImplementationDef

When objects are activated the BOA daemon needs to know certain essential information about the object. All such information is encapsulated in the `ImplementationDef` of the object. This

section discusses the `ImplementationDef` definition and how to set, get and modify the `ImplementationDef`.

`ImplementationDef` is defined in DCE IDL as follows:

```
[
uuid( 667F1C41-0E07-11CF-8706-0000C0284909 ),
version( 1.0 ),
pointer_default( ptr )
]
interface CORBA_ImplementationDef
{
  /* Activation Policy */
  enum ImplementationDef_Policy {
    SHARED_SERVER,
    UNSHARED_SERVER,
    /* SERVER_PER_METHOD - Not implemented! */
    /* PERSISTENT_SERVER - Not implemented! */
  };

  CORBA_ImplementationDef_ReferenceData;
  CORBA_ImplementationDef_ImplArgList;
  CORBA_ImplementationDef_ImplEnvList;

  // attribute String interfaceName;
  CORBA_String CORBA_ImplementationDef_get_interfaceName();
  CORBA_void CORBA_ImplementationDef_set_interfaceName(
    [in] CORBA_String _interfaceName );

  // attribute String objectName;
  CORBA_String ImplementationDef_get_objectName();
  CORBA_void CORBA_ImplementationDef_set_objectName(
    [in] CORBA_String _objectName );

  /* attribute String pathName; */
  CORBA_String CORBA_ImplementationDef_get_pathName();
  CORBA_void CORBA_ImplementationDef_set_pathName(
    [in] CORBA_String _pathName );

  /* attribute Policy policy; */
  CORBA_ImplementationDef_Policy CORBA_ImplementationDef_get_policy();
  CORBA_void CORBA_ImplementationDef_set_policy(
    [in] CORBA_ImplementationDef_Policy _policy );
```

```
  /* attribute ReferenceData id; */
  CORBA_ReferenceData CORBA_ImplementationDef_get_id();
  CORBA_void CORBA_ImplementationDef_set_id(
    [in] CORBA_ReferenceData _id );

  /* attribute ImplArgList args; */
  CORBA_ImplArgList CORBA_ImplementationDef_get_args();
  CORBA_void CORBA_ImplementationDef_set_args(
    [in] CORBA_ImplementationDef_ImplArgList _args );

  /* attribute ImplEnvList env; */
  CORBA_ImplementationDef_ImplEnvList CORBA_ImplementationDef_get_env();
  CORBA_void CORBA_ImplementationDef_set_env(
    [in] CORBA_ImplementationDef_ImplEnvList _env );
};
```

The data members of the `ImplementationDef` follow, accessor methods are provided to get and set the value of the data members :

- `interfaceName`. The interface name of the object. This name is provided when the object interface is defined in IDL.
- `objectName`. The name of the particular instance of the object. This name is given to the object by the person installing the object.
- `pathName`. Exact pathname of the executable for the object implementation. This path will be used at activation time to start up the binary. The file specified should have execute permission.
- `policy`. Activation policy for the object. This is one of the policies discussed above.
- `id`. This data is chosen by the implementation and is opaque to the ORB and the BOA. A typical implementation can use the id value to distinguish different objects.
- `args`. List of all arguments to be passed to the server when it is activated by the BOA daemon.
- `env`. List of all environment variables to be passed to the server when it is activated by the BOA daemon.

The `ImplementationDef` is supplied to the ORB when the object is installed using the `registerObject` command or created via `CORBA_BOA_create` described in section 7.2.2.

6.6 Summary

Dynamic applications are a significant differance between DCE and CORBA paradigms. This chapter has shown the use of the dynamic interface and how DCE can be used to implement it.

Chapter 7

Object Implementation

7.1 Introduction

The previous chapter discussed the notion of object names and how the client can bind to a specific object instance. CORBA provides a great degree of flexibility and control in terms of locating objects and using particular instances of objects. This chapter discusses CORBA's object location scheme and the bind process in greater detail.

An object adapter registers the application's interfaces, instantiates new objects, gives them unique IDs, advertises their existence, invokes their methods when clients request it, and manages concurrent requests for their services. CORBA specifies a basic object adapter (BOA) which is required by every ORB; an Object Implementation may choose a different OA based on what kind of services the Object Implementation requires. Object Adapters may :

- Register server classes with the Implementation Repository which can be regarded as a persistent store managed by the OA.
- Instantiate new objects at runtime. The number of instances created is a function of demand for services. The adapter is responsible for balancing the supply of objects with the incoming client demands.
- Generate and manage object references. The OA assigns unique reference IDs to the new objects it creates.
- Broadcast the presence of the object servers. The OA may broadcast the services it provides on the ORB, or it may respond to queries from the ORB code. Either way, its function is to tell the outside world about the services it manages.
- Handle incoming client calls. The object adapter interacts with the top messáging layer of the ORB communication stack, peels off the request, and hands it to the interface skeleton. The skeleton interprets the incoming parameters and presents them in a form that is acceptable to the object methods invocation.

- Route the upcall to the appropriate method. The OA is implicitly involved in the invocation of the methods described in the skeleton stubs. It may be involved in activating the implementation and authenticating incoming requests.

7.2 Object Activation

CORBA provides automatic activation of servers. In this case, a client can initiate a request to a specific object. CORBA with the help of the BOA daemon will determine if the object is already active and running. If the object is active, then the client can start communication with the object. If the object is not active, that is if no server embedding that object is running, the BOA daemon will automatically start up that object implementation. Basically, the BOA daemon queries the Implementation Repository to retrieve information on which server defines the object, and then activates the appropriate server, if found. The object implementation will perform some initialisation [12]. Once this is done, the client will be able to invoke requests on that object.

To use the BOA daemon, it is necessary to start the daemon on the host where the objects would be activated.

7.2.1 Server Activation Policy

According to the CORBA specification [9, section 9.2.2], servers can have different activation policies. The activation policy describes the rules that a given implementation follows when multiple objects or implementations are active. Activation policies only apply to persistent objects. There are four activation policies that CORBA's BOA specifications support, but only the shared and unshared server policies are implemented by my ORB.

- *Shared Server Policy* where multiple active objects of a given implementation share the same server.
- *Unshared Server Policy* where only one object of a given implementation at a time can be active in one server.
- *Server-per-method Policy* where each invocation of a method is implemented by a separate server being started, with the server terminating when the method completes.
- *Persistent Server Policy* where the server is activated by something outside the BOA. The server nonetheless must register with the BOA to receive invocations. A persistent server is assumed to be shared by multiple active servers.

The activation policy of a server is specified when the server is installed, that is when it is registered with the BOA daemon.

The `ImplementationDef` is supplied to the ORB when the object is installed using the `registerObject` command or created via `CORBA_BOA_create`.

The `ImplementationDef` of an object can be changed by the object implementation using

```
[
uuid( B87BFCC1-0E11-11CF-9126-0000C0284909 ),
version( 1.0 ),
pointer_default( ptr )
]
interface CORBA_BOA
{

    ...

  CORBA_void CORBA_BOA_change_implementation(
    [in] CORBA_Object obj,
    [in] CORBA_ImplementationDef impl );

    ...

}
```

`CORBA_BOA_changeImplementation()` can be used for instance to change the pathname of the object implementation.

7.2.2 Registering Objects with the BOA

The `registerObject` or `CORBA_BOA_create()` is used to register an object with the BOA daemon.

• **Registering Objects using** `registerObject`

`registerObject` registers the object with the BOA daemon and adds the object to the Implementation Repository. The `registerObject` command is invoked with the following arguments :

```
registerObject interfaceName objectName pathName
               [-p policy] [-a arguments] [-e environment]
```

- `interfaceName`. The interface name of this object.
- `objectName`. The name of this instance of the object.
- `pathName`. The name of the executable for the object implementation.
- `policy`. The policy for this server. Shared Server policy is the default.
- `arguments`. List of all arguments to be passed to the server when it is started.
- `environment`. List of all environment variables to be passed to the server when it is started.

• **Registering Objects using** `CORBA_BOA_create()`

`CORBA_BOA_create()` is an operation provided on the BOA. `CORBA_BOA_create()` provides a programmatic interface to the `registerObject` command above. As a result of calling `CORBA_BOA_create()`, an ORB object is created and registered with the BOA daemon. As in `registerObject`, the server which includes this object can be automatically activated by the BOA daemon when a client invokes a request on that object and the server is not active.

7.2.3 Unregistering Objects with the BOA

When the objects are no longer supported, or the service is temporarily unavailable the object should be unregistered with the BOA daemon.

• **Unregistering Objects using** `unregisterObject`

`unregisterObject` unregisters the object with BOA daemon and removes this object's entry from the Implementation Repository. As a result this object will no longer be available to clients. Similarly any attempts to perform a `CORBA_BOA_changeImplementation()` on this object will fail.

• **Unregistering Objects using** `CORBA_BOA_dispose()`

Invoking `CORBA_BOA_dispose()` by passing it a reference to the object to be deleted will cause it to be unregistered with the BOA daemon. Again, clients will no longer be able to access this object.

7.2.4 Object Initialization upon Activation

It is necessary to inform the BOA that it is ready to service requests before the BOA can deliver any requests to the BOA.

- **Shared Server**
  Using a Shared Server policy, a single server is capable of implementing multiple objects. Once the server has initialised itself, it calls `CORBA_BOA_implIsReady()` on the BOA. Subsequently, the BOA will start delivering requests to the object in that server. The server remains active and will receive requests until it calls `CORBA_BOA_deactivateImpl()`
- **Unshared Server**
  Using a Shared Server policy, each object is implemented on a single server. Once the server has initialised itself, it notifies the BOA via the `CORBA_BOA_objIsReady()` method. Subsequently the BOA will deliver requests to the object until the `CORBA_BOA_deactivateObj()`

```
[
uuid( F0908561-0E41-11CF-919D-0000C0284909 ),
```

```
version( 1.0 )
]
interface CORBA_BOA
{

  ...

  /* Shared Server Policy */
  CORBA_void CORBA_BOA_implIsReady();

  CORBA_void CORBA_BOA_deactivateImpl(
    [in] CORBA_ImplementationDef impl );

  /* Unshared Server Policy */
  CORBA_void CORBA_BOA_objIsReady(
    ]in] CORBA_Object obj,
    [in[ CORBA_ImplementationDef impl );

  CORBA_void CORBA_BOA_deactivateObj(
    [in] CORBA_Object impl );

  ...

}
```

7.3 Object Request Broker

To centralise the management of the ORB, all clients and implementations can communicate with a server whose job it is to route requests from clients to implementations. The routing of the requests is done by DCE. Basically, when a request arrives, DCE queries the naming service to find which server will service this request and the request is then routed to the appropriate server.

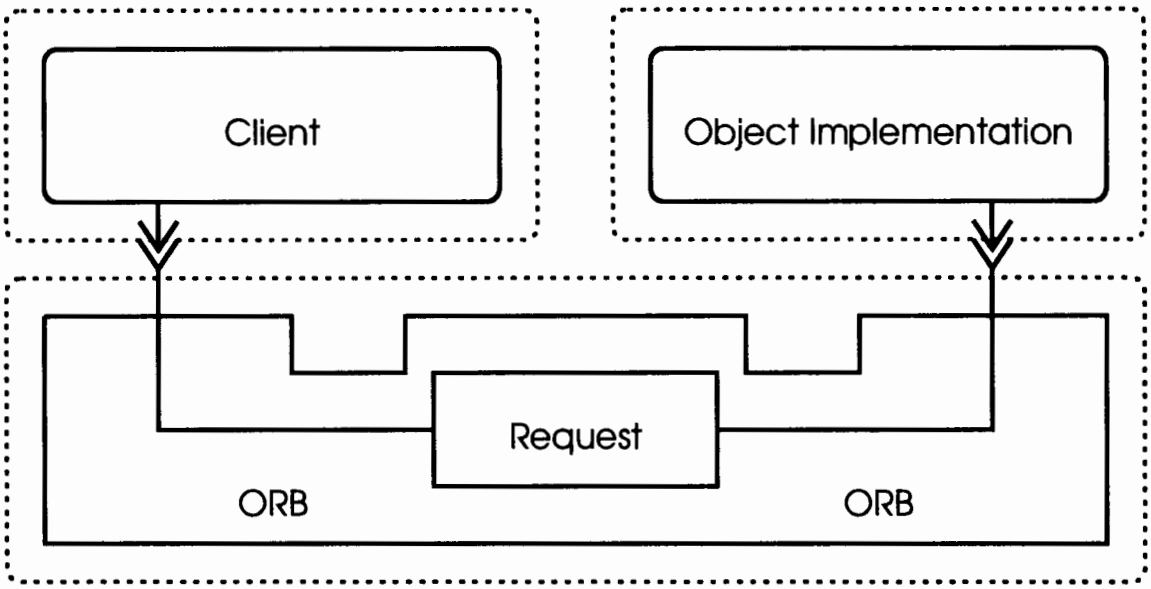


Figure 15: Server Based Object Request Broker

Chapter 8

Case Studies

8.1 The "Stack" Application

A "stack" application is presented to demonstrate that after very little work the stack class can be used to implement distributed objects that are accessed remotely. The example first presents the *"stack"* application components and the steps that the programmer must perform before the application can be run. Finally, I include a benchmark program to test the performance of my ORB and a description of the runtime activity that results from executing the application.

8.1.1 Interface Definition Language

The interface for the *"stack"* class is defined in a CORBA IDL file called `stack.idl"`.

```
interface stack
{
  const long stackSize = 100;
  boolean full();
  boolean empty();
  long top();
  void pop();
  void push( in long element );
};
```

Client program using a remote stack

A simple client program written to create and access a remote "stack" object is displayed below. The exact location of the object does not matter to the application.

```
#include "stack_CStub.h"

int main()
{
  stack _stk;
  CORBA_Environment *_eNv;

  CORBA_ORB_init( &_eNv );
  _stk = stack_bind( _eNv );

  /* Note that the stack is accessed as if it is local */
  stack_push( _stk,100,&_eNv );
  stack_push( _stk,200,&_eNv );
  stack_pop( _stk,&_eNv );

  if( !stack_empty( _stk,&_eNv ))
    printf( "Top: %d\n",stack_top( _stk,_eNv );

  CORBA_ORB_uninit( &_eNv );
};
```

Server program

The server program defines all functions of the `"stack"` interface.

```
struct stack{
  CORBA_Long _currentSize;
  CORBA_Long _stackArray[stackSize];
} stack;

CORBA_Boolean stack_full(
  stack _o,
  CORBA_Environment *_eNv )
{
  if( _o._currentSize >= stackSize )
    return TRUE;
  else
    return FALSE;
};

CORBA_Boolean stack_empty(
  stack _o,
```

```
  CORBA_Environment *_eNv )
{
  if( _o._currentSize == 0 )
    return TRUE;
  else
    return FALSE;
};

CORBA_Long stack_top(
  stack _o,
  CORBA_Environment *_eNv )
{
  return _o._stackArray[_o._currentSize - 1];
};

CORBA_Void stack_pop(
  stack _o,
  CORBA_Environment *_eNv )
{
  return _o._stackArray[--_o._currentSize ];
};

CORBA_Void stack_push(
  stack _o,
  CORBA_Long element,
  CORBA_Environment *_eNv )
{
  _o._stackArray[ _o._currentSize++] = element;
};

main()
{
  CORBA_Environment _eNv;

  /* Initialize the ORB runtime environment */
  CORBA_ORB_init();

  /* BOA ready to receive requests */
  CORBA_BOA_impl_is_ready();

  CORBA_ORB_uninit();
};
```

In the `main()` routine the server initializes the objects with the ORB and indicates that the server is ready to receive requests.

8.1.2 Binding Files Generated

The CORBA IDL file is compiled and the following files are generated. Both DCE methods are shown as described in Chapter 5.

DCE Data Marshalling

The following simple stack interface defined in the file `"stack.idl"`.

DCE IDL File

The *"stack_DCE.idl"* file is generated.

```
[
uuid( 2E9A5A41-1DFE-11CF-AB61-0000C0284909 ),
version( 1.0 )
]
interface stack_DCE
{
  /* ... various CORBA definitions ... */

  /* interface stack */
  typedef CORBA_Object stack;
  typedef stack *stack_Ptr;

  /* DSI Prototypes */
  void stack_full(
    [in] stack _o,
    [out] CORBA_Environment *_eNv );

  void stack_empty(
    [in] stack _o,
    [out] CORBA_Environment *_eNv );

  void stack_top(
    [in] stack _o,
    [out] CORBA_Environment *_eNv );

  void stack_pop(
```

```
    [in] stack _o,
    [out] CORBA_Environment *_eNv );

  void stack_push(
    [in] stack _o,
    [in] CORBA_long _element;
    [out] CORBA_Environment *_eNv );
};
```

C Data Type Header File
The *"stack_IDLTypes.h"* file is generated.

```
#ifndef _STACK_IDLTYPES_H_
#define _STACK_IDLTYPES_H_

/* interface stack */
#define stack_stackSize 100
#define stack_objUUID 70875521-1D45-11CF-9095-0000C0284909;

#endif /* _STACK_IDLTYPES_H_ */
```

The "Client" Side
A *The Client C Stub Header File* is generated

```
#ifndef _STACK_CSTUB_H_
#define _STACK_CSTUB_H_
/* "stack_CStub.h" */

#include "CORBA.h"
#include "stack_IDLTypes.h"
#include "stack_DCE.h"

/* DII Prototypes */

#endif /* _STACK_CSTUB_H_ */
```

The "Server" Side
A *The Server C Skeleton Header File* is generated

```
#ifndef _STACK_SSKELETON_H_
#define _STACK_SSKELETON_H_
```

```
/* "stack_SSkeleton.h" */

#include "CORBA.h"
#include "stack_IDLTypes.h"
#include "stack_DCE.h"

/* DSI Prototypes defined in "stack_DCE.h" */

#endif /* _STACK_SSKELETON_H_ */
```

UNO DCE Mapping

The following files are generated which use the DCE pipe interface. *Note* that no DCE files are generated.

C Data Type Header File
The "*stack_IDLTypes.h*" file is generated.

```
#ifndef _STACK_IDLTYPES_H_
#define _STACK_IDLTYPES_H_

#define stack_stackSize 100

/* interface stack */
typedef CORBA_Object stack;
typedef stack *stack_ptr;

#define stack_objUUID "B5BD5EC1-346D-11CF-90A6-0000C0284909"

#endif /* _STACK_IDLTYPES_H_ */
```

The "Client" Side
The Client C Stub Header File

```
#ifndef _STACK_CSTUB_H_
#define _STACK_CSTUB_H_
/* "stack_CStub.h" */

#include "CORBA.h"
#include "stack_IDLTypes.h"
#include "stack_DCE.h"

/* DII Prototypes */
```

```
CORBA_Boolean stack_full( stack,CORBA_Environment * );
CORBA_Boolean stack_empty( stack,CORBA_Environment * );
CORBA_Long stack_top( stack,CORBA_Environment * );
void stack_pop( stack,CORBA_Environment * );
void stack_push( stack,long,CORBA_Environment * );

#endif /* _STACK_CSTUB_H_ */
```

The Client C Stub File

Only comments are indicated in the stubs where the data marshalling/unmarshalling takes place to reduce the amount of bulk code in this section.

```
/* "stack_CStub.c" */

#include "stack_CStub.h"

CORBA_Boolean stack_full(
  stack _o,
  CORBA_Environment *_eNv )
{
  /* data marshall in,inout parameters */
  _invoke( _ORB_globalBindingHandle,inData,outData );
  /* data unmarshall inout,out parameters */
};

CORBA_Boolean stack_empty(
  stack _o,
  CORBA_Environment *_eNv )
{
  /* data marshall in,inout parameters */
  _invoke( _ORB_globalBindingHandle,inData,outData );
  /* data unmarshall inout,out parameters */
};

CORBA_Long stack_top(
  stack _o,
  CORBA_Environment *_eNv )
{
  /* data marshall in,inout parameters */
  _invoke( _ORB_globalBindingHandle,inData,outData );
  /* data unmarshall inout,out parameters */
};
```

```
void stack_pop(
  stack _o,
  CORBA_Environment *_eNv )
{
  /* data marshall in,inout parameters */
  _invoke( _ORB_globalBindingHandle,inData,outData );
  /* data unmarshall inout,out parameters */
};

void stack_push(
  stack _o,
  long _element,
  CORBA_Environment *_eNv )
{
  /* data marshall in,inout parameters */
  _invoke( _ORB_globalBindingHandle,inData,outData );
  /* data unmarshall inout,out parameters */
};
```

The "Server" Side

The Server C Skeleton Header File

```
#ifndef _STACK_SSKELETON_H_
#define _STACK_SSKELETON_H_

/* "stack_SSkeleton.h" */

#include "CORBA.h"
#include "stack_IDLTypes.h"
#include "stack_DCE.h"

/* DSI Prototypes defined in "stack_DCE.h" */

#endif /* _STACK_SSKELETON_H_ */

/* "stack_SSkeleton.c" */

#include "stack_SSkeleton.h"

void stack_full_SSkeleton(
```

```
  CORBA_Object _target,
  CORBA_ServerRequest *_request,
  CORBA_Environment *_eNv )
{
  stack _o;
  CORBA_Boolean _result;
  /* data unmarshall _request */
  _result = stack_full( _o,_eNv );
  /* data marshall _request */
};

void stack_empty_SSkeleton(
  CORBA_Object _target,
  CORBA_ServerRequest *_request,
  CORBA_Environment *_eNv )
{
  stack _o;
  CORBA_Boolean _result;
  /* data unmarshall _request */
  _result = stack_empty( _o,_eNv );
  /* data marshall _request */
};

void stack_top_SSkeleton(
  CORBA_Object _target,
  CORBA_ServerRequest *_request,
  CORBA_Environment *_eNv )
{
  stack _o;
  /* data unmarshall _request */
  _result = stack_top( _o,_eNv );
  /* data marshall _request */
};

void stack_pop_SSkeleton(
  CORBA_Object _target,
  CORBA_ServerRequest *_request,
  CORBA_Environment *_eNv )
{
  stack _o;
  /* data unmarshall _request */
  _result = stack_pop( _o,_eNv );
```

```
  /* data marshall _request */
};

void stack_push_SSkeleton(
  CORBA_Object _target,
  CORBA_ServerRequest *_request,
  CORBA_Environment *_eNv )
{
  stack _o;
  long _element;
  /* data unmarshall _request */
  _result = stack_push( _o,_element,_eNv );
  /* data marshall _request */
};
```

8.1.3 Dynamic Invocation Interface

A simple client program written to create and access a remote "stack" object using the DII is displayed below. Interface definition information is inserted into the Interface Repository when the CORBA IDL file is compiled.

```
#include "CORBA.h"

main()
{
  stack _stk;
  CORBA_Environment *_eNv;

  CORBA_DII_Request *_reQuest;
  CORBA_NVList *_argList;
  CORBA_Identifier _name;
  CORBA_Long _length;
  CORBA_Long _value1,_result;
  CORBA_OperationDef _opDef;
  CORBA_OperationDescription *_opDesc;
  CORBA_Contained_Description _desc;
  CORBA_TypeCode _tc;

  CORBA_ORB_init( &_eNv );

  /* Equivalent IDL request - stack_push( _stk,100,&_eNv ); */
```

```
/* Get the OperationDef from the Interface Repository */
_opDef = CORBA_Repository_lookup_id( CORBA_InterfaceRepository_Obj,
                                     "stack_push",
                                     &_eNv );

/* Create a NamedValue list for the operation */
CORBA_ORB_create_operation_list( CORBA_ORB_Obj,_opDef,&_argList );

/* Insert argument 1 info into _argList */
_value1 = 100;
CORBA_ORB_get_arg( argList,&_tc,&_name,0,&_length,&_flags,&_eNv );
CORBA_ORB_add_arg( argList,_tc,&_value1,sizeof( CORBA_Long ),ARG_IN,&_eNv );

/* Get the operation description structure */
_desc = CORBA_OperationDef_describe( _opDef,&_eNv );
_opDesc = ( CORBA_OperationDescription * )_desc.value._value;

/* Fill in the TypeCode field for the result */
_result.argument._type = _opDesc->result;

/* Create the request, CORBA_DII_Request_Obj */
_reQuest = CORBA_DII_Request_create_request(
  _stk,"stack_push",_argList,_resultNV,&CORBA_DII_Request_Obj,
  ( CORBA_Context * )NULL,&_eNv );

/* Finally, invoke the request */
_reQuest = CORBA_DII_Request_invoke( CORBA_DII_Request_Obj,
                                     ( CORBA_Flags )0,
                                     &_eNv );

/* Equivalent IDL request - stack_push( _stk,200,&_eNv ); */

/* Create a NamedValue list for the operation */
CORBA_ORB_create_operation_list( CORBA_ORB_Obj,_opDef,&_argList );

/* Insert argument 1 info into _argList */
_value1 = 200;
CORBA_ORB_get_arg( argList,&_tc,&_name,0,&_length,&_flags,&_eNv );
CORBA_ORB_add_arg( argList,_tc,&_value1,sizeof( CORBA_Long ),ARG_IN,&_eNv );

/* Get the operation description structure */
_desc = CORBA_OperationDef_describe( _opDef,&_eNv );
```

```
_opDesc = ( CORBA_OperationDescription * )_desc.value._value;

/* Fill in the TypeCode field for the result */
_result.argument._type = _opDesc->result;

/* Create the request, CORBA_DII_Request_Obj */
_reQuest = CORBA_DII_Request_create_request(
  _stk,"stack_push",_argList,_result,&CORBA_DII_Request_Obj,
  ( CORBA_Context * )NULL,&_eNv );

/* Finally, invoke the request */
_reQuest = CORBA_DII_Request_invoke( CORBA_DII_Request_Obj,
                                     ( CORBA_Flags )0,
                                     &_eNv );

/* Equivalent IDL request - stack_pop( _stk,&_eNv ); */

/* Get the OperationDef from the Interface Repository */
_opDef = CORBA_Repository_lookup_id( CORBA_InterfaceRepository_Obj,
                                     "stack_pop",
                                     &_eNv );

/* Create a NamedValue list for the operation */
_argList = ( CORBA_NVList * )NULL;

/* Get the operation description structure */
_desc = CORBA_OperationDef_describe( _opDef,&_eNv );
_opDesc = ( CORBA_OperationDescription * )_desc.value._value;

/* Fill in the TypeCode field for the result */
_result.argument._type = _opDesc->result;

/* Create the request, CORBA_DII_Request_Obj */
_reQuest = CORBA_DII_Request_create_request(
  _stk,"stack_push",_argList,_result,&CORBA_DII_Request_Obj,
  ( CORBA_Context * )NULL,&_eNv );

/* Finally, invoke the request */
_reQuest = CORBA_DII_Request_invoke( CORBA_DII_Request_Obj,
                                     ( CORBA_Flags )0,
                                     &_eNv );
```

```
/* Equivalent IDL Request - if( !stack_empty( _stk,&_eNv )) */

/* Get the OperationDef from the Interface Repository */
_opDef = CORBA_Repository_lookup_id( CORBA_InterfaceRepository_Obj,
                                     "stack_empty",
                                     &_eNv );

/* Create a NamedValue list for the operation */
_argList = ( CORBA_NVList * )NULL;

/* Get the operation description structure */
_desc = CORBA_OperationDef_describe( _opDef,&_eNv );
_opDesc = ( CORBA_OperationDescription * )_desc.value._value;

/* Fill in the TypeCode field for the result */
_result.argument._type = _opDesc->result;

/* Create the request, CORBA_DII_Request_Obj */
_reQuest = CORBA_DII_Request_create_request(
  _stk,"stack_push",_argList,_resultNV,&CORBA_DII_Request_Obj,
  ( CORBA_Context * )NULL,&_eNv );

/* Finally, invoke the request */
_reQuest = CORBA_DII_Request_invoke( CORBA_DII_Request_Obj,
                                     ( CORBA_Flags )0,
                                     &_eNv );

if( !*( CORBA_Boolean * )( _reQuest->result.argument._value )){

  /* Equivalent IDL Request - printf( "Top: %d\n",stack_top( _stk,_eNv ); */

  /* Get the OperationDef from the Interface Repository */
  _opDef = CORBA_Repository_lookup_id( CORBA_InterfaceRepository_Obj,
                                       "stack_empty",
                                       &_eNv );

  /* Create a NamedValue list for the operation */
  _argList = ( CORBA_NVList * )NULL;

  /* Get the operation description structure */
  _desc = CORBA_OperationDef_describe( _opDef,&_eNv );
  _opDesc = ( CORBA_OperationDescription * )_desc.value._value;
```

```
    /* Fill in the TypeCode field for the result */
    _result.argument._type = _opDesc->result;

    /* Create the request, CORBA_DII_Request_Obj */
    _reQuest = CORBA_DII_Request_create_request(
      _stk,"stack_push",_argList,_resultNV,&CORBA_DII_Request_Obj,
      ( CORBA_Context * )NULL,&_eNv );

    /* Finally, invoke the request */
    _reQuest = CORBA_DII_Request_invoke( CORBA_DII_Request_Obj,
                                         ( CORBA_Flags )0,
                                         &_eNv );

    printf( "Top: %d\n", *( CORBA_Long * )_reQuest->result.argument._value );

  }

  CORBA_ORB_uninit( &_eNv );
};
```

8.2 Performance Test

The following benchmark stresses most of the features available in CORBA IDL. It begins by timing a series of *oneway* calls, that is calls for which no response is expected. These calls pass a variety of parameters covering all the major datatypes in CORBA IDL using only the parameter passing mode `in`.

The benchmark also times the *synchronous* calls in CORBA, that is calls for which a response and data is expected. Again, these tests pass a variety of CORBA datatypes as parameters using all the parameter passing modes, namely `in`, `out` and `inout`.

The source for this benchmark was provided as one of the standard demonstration programs from Postmodern Computing Inc.

The IDL for the performance benchmark is given below. My tests were carried out on one machine : An NCR3450 (4 486DX50 processors with 196MB of memory) running NCR MP-RAS 02.02 (System V.4) operating system. This is the only machine on which we could have DCE installed. Due to the fact that we were limited to only one machine running DCE, performance results can only be compared to the commercial ORBs results where the server object and client are separate processes on the same machine.

Note : The timings are all measured in milliseconds.

8.2.1 Interface Definition Language

The interfaces for the "Oneway" and "RequestReply" classes are defined in CORBA IDL as follows :

```
/* This IDL is provided coutesy of Postmodern Computing Inc.
** IDLs to do performance testing of oneway and request/reply interfaces
** These tests also include arguments of various types
*/

typedef sequence<short> shortSeq;
typedef sequence<long> longSeq;
typedef sequence<float> floatSeq;
typedef sequence<double> doubleSeq;
typedef sequence<octet> octetSeq;
typedef sequence<string> stringSeq;
typedef sequence<char> charSeq;

struct PerfStruct {
  short shortVal;
  long longVal;
  float floatVal;
  double doubleVal;
  char charVal;
  string stringVal;
};

typedef sequence<PerfStruct> structSeq;
typedef PerfStruct structArray[100];

interface Oneway
{
  oneway void test_no_param();
  oneway void test_prim_args(
    in short shortVal,in long longVal,in float floatVal,
    in double doubleVal,in char charVal,in string stringVal );
  oneway void test_struct( in PerfStruct structVal );
  oneway void test_prim_seq(
    in shortSeq shortVal,in longSeq longVal,in floatSeq floatVal,
    in doubleSeq doubleVal,in charSeq charVal,in stringSeq stringVal );
  oneway void test_struct_seq( in structSeq structVal );
  oneway void test_struct_array( in structArray arrayVal );
```

```
};

interface RequestReply
{
  long test_prim_args(
    in short shortVal,in long longVal,in float floatVal,
    in double doubleVal,in char charVal,in string stringVal,
    inout short inoutShort,inout long inoutLong,inout float inoutFloat,
    inout double inoutDouble,inout char inoutChar,inout string inoutString,
    out short outShort,out long outLong,out float outFloat,
    out double outDouble,out char outChar, out string outString );
  long test_struct_args(
    in PerfStruct structVal,
    inout PerfStruct inoutStruct,
    out PerfStruct outStruct );
  long test_prim_seq(
    in shortSeq shortVal,in longSeq longVal,in floatSeq floatVal,
    in doubleSeq doubleVal,in charSeq charVal,in stringSeq stringVal,
    inout shortSeq inoutShort,inout longSeq inoutLong,
    inout floatSeq inoutFloat,inout doubleSeq inoutDouble,
    inout charSeq inoutChar,inout stringSeq inoutString,
    out shortSeq outShort,out longSeq outLong,out floatSeq outFloat,
    out doubleSeq outDouble,out charSeq outChar,out stringSeq outString );
  long test_struct_seq(
    in structSeq structVal,
    inout structSeq inoutStruct,
    out structSeq outStruct );
  long test_struct_array(
    in structArray structVal,
    inout structArray inoutStruct,
    out structArray outStruct );
};
```

8.2.2 Using the CORBA IDL Compiler

The hardware used for the performance results for ORBeline (See Section 2.6.1) was a SUN Sparc 5 (32MB) running Solaris 2.3. The tests for Iona's Orbix 1.3 were carried out on three machines : A SUN Sparc 20 (64MB) running Solaris 2.3, a SUN Sparc Classic (32MB) running Solaris 2.3 and a HP 9000/712 (32MB) running HPUX 9.0.

8.2.3 Using the Dynamic Invocation Interface

	Bytes Sent/Received	NCR3450 DCE Data Marshalling	NCR3450 UNO DCE Mapping
Oneway_test_no_param	0	33.45	13.34
Oneway_test_prim_args	25	33.62	13.87
Oneway_test_struct	25	31.10	14.10
Oneway_test_prim_seq	2500	32.22	14.22
Oneway_test_struct_seq	2500	36.19	16.60
Oneway_test_struct_array	2500	40.12	18.01
RequestReply_test_prim_args	100	42.22	20.45
RequestReply_test_struct_args	100	43.01	20.48
RequestReply_test_prim_seq	10000	50.15	40.34
RequestReply_test_struct_seq	10000	63.95	43.43
RequestReply_test_struct_array	10000	60.91	42.03

Table 6: Times for CORBA IDL Compiler using both DCE Data Marshalling and UNO Pipe-based DCE Mapping

	Bytes Sent/Received	ORBeline Sparc 5	Orbix Sparc 20	Orbix Sparc Classic	Orbix HP 9000/712
Oneway_test_no_param	0	0.00028	0.211	0.337	0.649
Oneway_test_prim_args	25	0.00044	0.240	0.396	0.387
Oneway_test_struct	25	0.00028	0.240	0.396	0.387
Oneway_test_prim_seq	2500	0.00038	1.195	1.231	1.768
Oneway_test_struct_seq	2500	0.00028	1.811	3.011	3.991
Oneway_test_struct_array	2500	0.00032	1.791	3.023	4.159
RequestReply_test_prim_args	100	0.01980	2.07	3.496	5.294
RequestReply_test_struct_args	100	0.02844	2.156	3.48	5.259
RequestReply_test_prim_seq	10000	0.00415	11.135	18.471	27.007
RequestReply_test_struct_seq	10000	0.00253	16.481	35.563	48.557
RequestReply_test_struct_array	10000	0.00256	14.791	31.134	40.11

Table 7: Times for CORBA IDL Compiler using ORBeline and Orbix

The hardware used for the performance results for ORBeline (See Section 2.6.1) was a SUN Sparc 5 (32MB) running Solaris 2.3. Results were unavailable for Iona's Orbix 1.3.

8.2.4 General Observations

Although my performance results seem slower than those achieved by ORBeline and Orbix it is nearly impossible to compare my results with those of the commercial products. The reason is that the NCR3450 machine used is an Intel 486DX50 processor which is much slower in comparison to those used by ORBeline and Orbix. The results should rather be compared with these obtained on similar machines using different ORBs.

NCR3450 is a multiprocessor machine, which allows for more work but does not increase the processing power of the machine. Hence, DCE will only run on a single processor at a time. Also, our NCR3450 machine is a central server for all second and third year undergraduate

	Bytes Sent/Received	NCR3450 DII
Oneway_test_no_param	0	38.45
Oneway_test_prim_args	25	43.62
Oneway_test_struct	25	43.10
Oneway_test_prim_seq	2500	42.22
Oneway_test_struct_seq	2500	56.19
Oneway_test_struct_array	2500	53.12
RequestReply_test_prim_args	100	42.22
RequestReply_test_struct_args	100	43.01
RequestReply_test_prim_seq	10000	55.15
RequestReply_test_struct_seq	10000	67.95
RequestReply_test_struct_array	10000	60.91

Table 8: Times for Dynamic Invocation Interface

	Bytes Sent/Received	ORBeline DII
Oneway_test_no_param	0	0.31461
Oneway_test_prim_args	25	0.37577
Oneway_test_struct	25	0.57052
Oneway_test_prim_seq	2500	7.457
Oneway_test_struct_seq	2500	11.2361
Oneway_test_struct_array	2500	7.84424
RequestReply_test_prim_args	100	2.83452
RequestReply_test_struct_args	100	2.79522
RequestReply_test_prim_seq	10000	22.0547
RequestReply_test_struct_seq	10000	30.9805
RequestReply_test_struct_array	10000	23.3143

Table 9: Times for Dynamic Invocation Interface using ORBeline

students. This means that although run during times when no other students were working its performance is affected by many other application processes running. In contrast, it is probable that the results for the commercial ORBs were tested on *standalone machines* that is, machines which are not running any other processes.

Furthermore we are using an old operating system compared with those used for the commercial ORBs, and StarPro DCE 1.00.00 which is an old version of DCE. This will affect the performance of our ORB.

However, if you look across the tables the performance results show a consistent increase in time with an increase in the amount of data sent over the network. This is consistent with the relative times of the commercial ORBs. Also, if the times for the two versions of the CORBA IDL compiler are compared there seems to be a speedup for the UNO Pipe Based DCE based IDL Compiler (See Table 6). Where DCE RPC pipes are used for large volumes of data one would expect a speedup in transmission.

Loss in performance can be attributed mainly to the overhead of adding object oriented features to DCE.

Chapter 9

Conclusions

Distributed computing represents the direction in which the mainstream of business computing is likely to evolve. Object Request Brokers are an evolutionary transition from client/server computing using the features of object orientation. The purpose of the research presented in this thesis was the design and implementation of a front end processor for CORBA IDL which produces equivalent data and operation definitions in DCE IDL. The features of the ORB developed has been described in detail. Two mappings were produced to

- comply with DCE-CIOP interface, and
- use DCE IDL's data marshalling/unmarshalling.

9.1 Summary of Performance Results

The research work undertaken in this thesis entailed the investigation of the operation of a CORBA compliant ORB and, in particular, the performance which can be achieved by a DCE-based Object Request Broker which is CORBA compliant. It was found that the relative results were consistent across an increasing load over the network.

Although the performance results were slower than those achieved by other commercial Object Request Broker, that is ORBeline and Orbix, the ultimate gain would be the security services that are to be adopted by the Object Management Group. Loss in performance was mainly attributed to the overhead of adding object oriented features to DCE.

9.2 Future Work

Future work requires a CORBA IDL compiler which maps CORBA IDL to DCE using C++. It is also expected that my ORB, using DCE as the communication mechanism, will use the authentication and access control services currently available with DCE which will hopefully be

available from OMG in the near future. The Object Request Broker core could be extended to provide services such as security, transaction control, persistence and event-driven programming. The project provides a basis for further research. It is also hoped to test the performance results over a network of computers.

Appendix A

CORBA Interface Definition Language (IDL)

```
specification : definition+

definition : typeDcl ;
           | constDcl ;
           | exceptDcl ;
           | interface ;
           | module ;

module : module identifier { definition+ }

interface : interfaceDcl
          | forwardDcl

interfaceDcl : interfaceHeader { interfaceBody }

forwardDcl : interface identifier

interfaceHeader : interface identifier inheritanceSpec

interfaceBody : export*

export : typeDcl ;
       | constDcl ;
       | exceptDcl ;
       | attrDcl ;
```

```
        | opDcl ;

inheritanceSpec  : : scopedName {, scopedName }*

scopedName : identifier
           | :: identifier
           | scopedName :: identifier

constDcl : const constType identifier = constExpr

constType : integerType
          | charType
          | booleanType
          | floatingPtType
          | stringType
          | scopedName

constExpr : orExpr

orExpr : xorExpr
       | orExpr
       | xorExpr

xorExpr : andExpr
        | xorExpr ^ andExpr

andExpr : shiftExpr
        | andExpr & shiftExpr

shiftExpr : addExpr
          | shiftExpr >> addExpr
          | shiftExpr << addExpr

addExpr : multExpr
        | addExpr + multExpr
        | addExpr - multExpr

multExpr : unaryExpr
         | multExpr * unaryExpr
         | multExpr / unaryExpr
         | multExpr % unaryExpr

unaryExpr : primaryExpr
```

```
          | unaryOperator primaryExpr

unaryOperator : +
              | -
              | ~

primaryExpr : scopedName
            | literal
            | ( constExpr )

literal : integerLiteral
        | stringLiteral
        | characterLiteral
        | floatingPtLiteral
        | booleanLiteral

positiveIntConst : constExpr

typeDcl : typedef typeDeclarator
        | structType
        | unionType
        | enumType

typeDeclarator : typeSpec declaratorS

typeSpec : simpleTypeSpec
         | constrTypeSpec

simpleTypeSpec : baseTypeSpec
               | templateTypeSpec
               | scopedName

baseTypeSpec : floatingPtType
             | integerType
             | charType
             | wideCharType
             | booleanType
             | octetType
             | anyType

templateTypeSpec : sequenceType
                 | stringType
```

```
                | wideStringType

constrTypeSpec : structType
               | unionType
               | enumType

declaratorS : declarator
            | declaratorS , declarator

declarator : simpleDeclarator
           | complexDeclarator

simpleDeclarator : identifier

complexDeclarator : arrayDeclarator

floatingPtType : double
               | float
               | long double

integerType : signedInt
            | unsignedInt

signedInt : signedLongInt
          | signedShortInt
          | signedLongLongInt

signedLongLongInt : long long

signedLongInt : long

signedShortInt : short

unsignedInt : unsignedLongInt
            | unsignedShortInt
```

```
            | unsignedLongLongInt

unsignedLongLongInt : unsigned long long

unsignedLongInt : unsigned long

unsignedShortInt : unsigned short

charType : char

wideCharType : wchar

booleanType : boolean

octetType : octet

anyType : any

structType : struct identifier { memberList }

memberList : member
           | memberList member

member :  typeSpec declaratorS ;

unionType : union identifier switch ( switchTypeSpec ) { switchBody }

switchTypeSpec : integerType
               | charType
               | octetType
               | booleanType
               | enumType
               | scopedName

switchBody : caseS

caseS : caseStatement
      | caseS caseStatement

caseStatement : caseLabelS elementSpec ;

caseLabelS : caseLabel
```

```
            | caseLabelS caseLabel

caseLabel : default :
          | case constExpr :

elementSpec : typeSpec declarator

enumType : enum identifier { enumeratorS }

enumeratorS : enumerator
            | enumeratorS , enumerator

enumerator : identifier

sequenceType : sequence < simpleTypeSpec , positiveIntConst >
             | sequence < simpleTypeSpec >

stringType : string < positiveIntConst >
           | string

wideStringType : wstring
               | wstring < positiveIntConst >

arrayDeclarator : identifier fixedArraySizeS

fixedArraySizeS : fixedArraySize
                | fixedArraySizeS fixedArraySize

fixedArraySize :  [ positiveIntConst ]

attrDcl :  [readonly] attribute paramTypeSpec simpleDeclaratorS

simpleDeclaratorS : simpleDeclarator
                  | simpleDeclaratorS simpleDeclarator

exceptDcl :  exception identifier { [memberList] }

opDcl : [ opAttribute ] opTypeSpec identifier parameterDclList
        [ raisesExpr ] [ contextExpr ]

opAttribute : oneway

opTypeSpec : paramTypeSpec
```

```
            | void

parameterDclList : ()
                 | ( paramDclS )

paramDclS : paramDcl
          | paramDclS , paramDcl

paramDcl : paramAttribute paramTypeSpec simpleDeclarator

paramAttribute : in
               | out
               | inout

raisesExpr : raises ( scopedNameS )

contextExpr :]  context ( stringLiteralS )

stringLiteralS : stringLiteral
               | stringLiteralS , stringLiteral

paramTypeSpec : baseTypeSpec
              | stringType
              | scopedName
```

Appendix B

CORBA IDL Interface Definitions

This appendix provides a few of CORBA IDL interface definitions for the CORBA module.

B.1 Request Interface Definition

The *Request* interface is defined as follows :

```
module CORBA
{
  interface DII
  {
    interface Request
    {
      invoke( Flags );
      add_args(
        in Identifier name,
        in TypeCode arg_type,
        void *value,
        long length,
        Flags arg_flags );
      delete();
      send( Flags );
      get_response( Flags );
    };
  };
};
```

B.2 Interface Repository Interface Definitions

These are the IDL definitions from Appendix A of [11].

```
module CORBA
{
  #pragma prefix "omg.org"

  typedef string Identifier;
  typedef string ScopedName;
  typedef string RepositoryId;


  enum DefinitionKind {
    dk_none, dk_all,
    dk_Attribute, dk_Constant, dk_Exception, dk_Interface,
    dk_Module, dk_Operation, dk_Typedef,
    dk_Alias, dk_Struct, dk_Union, dk_Enum,
    dk_Primitive, dk_String, dk_Sequence, dk_Array
  };


  interface IRObject
  {
    readonly attribute DefinitionKind def_kind;
    void destroy();
  };

  typedef string VersionSpec;

  interface Contained;
  interface Repository;
  interface Container;

  interface Contained : IRObject
  {
    attribute RepositoryId id;
    attribute Identifier name;
    attribute VersionSpec version;
    readonly attribute Container defined_in;
    readonly attribute ScopedName absolute_name;
    readonly attribute Repository containing_repository;

    struct Description {
      DefinitionKind kind;
      Any value;
```

```
  };
  Description describe();
  void move(
    in Container new_container,
    in Identifier new_name,
    in VersionSpec new_version );
};
interface ModuleDef;
interface ConstantDef;
interface IDLType;
interface StructDef;
interface UnionDef;
interface EnumDef;
interface AliasDef;
interface InterfaceDef;
typedef sequence <InterfaceDef> InterfaceDefSeq;
typedef sequence <Contained> ContainedSeq;
struct StructMember {
  Identifier name;
  TypeCode type;
  IDLType type_def;
};
typedef sequence <StructMember> StructMemberSeq;
struct UnionMember {
  Identifier name;
  Any label;
  TypeCode type;
  IDLType type_def;
};
typedef sequence <UnionMember> UnionMemberSeq;
typedef sequence <Identifier> EnumMemberSeq;
interface Container : IRObject
{
  Contained lookup( in ScopedName search_name);
  ContainedSeq contents(
    in DefinitionKind limit_type,
    in boolean exclude_inherited );
  ContainedSeq lookup_name(
```

```
    in Identifier search_name,
    in long levels_to_search,
    in DefinitionKind limit_type,
    in boolean exclude_inherited );

  struct Description
    Contained contained_object;
    DefinitionKind kind;
    Any value;
  ;

  typedef sequence<Description> DescriptionSeq;

  DescriptionSeq describe_contents(
    in DefinitionKind limit_type,
    in boolean exclude_inherited,
    in long max_returned_objs );

  ModuleDef create_module(
    in RepositoryId id,
    in Identifier name,
    in VersionSpec version );

  ConstantDef create_constant(
    in RepositoryId id,
    in Identifier name,
    in VersionSpec version,
    in IDLType type,
    in Any value );

   StructDef create_struct(
    in RepositoryId id,
    in Identifier name,
    in VersionSpec version,
    in StructMemberSeq members );

  UnionDef create_union(
    in RepositoryId id,
    in Identifier name,
    in VersionSpec version,
    in IDLType discriminator_type,
    in UnionMemberSeq members );

  EnumDef create_enum(
    in RepositoryId id,
    in Identifier name,
    in VersionSpec version,
```

```
    in EnumMemberSeq members );

  AliasDef create_alias(
    in RepositoryId id,
    in Identifier name,
    in VersionSpec version,
    in IDLType original_type );

  InterfaceDef create_interface(
    in RepositoryId id,
    in Identifier name,
    in VersionSpec version,
    in InterfaceDefSeq base_interfaces );   };

interface IDLType : IRObject
{
  readonly attribute TypeCode type;
};

interface PrimitiveDef;
interface StringDef;
interface SequenceDef;
interface ArrayDef;

enum PrimitiveKind {
  pk_null,pk_void,pk_short,pk_long,pk_ushort,pk_ulong,
  pk_float,pk_double,pk_boolean,pk_char,pk_octet,
  pk_any,pk_TypeCode,pk_Principal,pk_string,pk_objref   };

interface Repository : Container
{
  Contained lookup_id( in RepositoryId search_id );

  PrimitiveDef get_primitive( in PrimitiveKind kind );

  StringDef create_string( in unsigned long bound );

  SequenceDef create_sequence(
    in unsigned long bound,
    in IDLType element_type );

  ArrayDef create_array(
    in unsigned long length,
    in IDLType element_type );
};

interface ModuleDef : Container,Contained {};

struct ModuleDescription {
  Identifier name;
```

```
  RepositoryId id;
  RepositoryId defined_in;
  VersionSpec version;
};

interface ConstantDef : Contained
{
  readonly attribute TypeCode type;
  attribute IDLType type_def;
  attribute Any value;
};

struct ConstantDescription {
  Identifier name;
  RepositoryId id;
  RepositoryId defined_in;
  VersionSpec version;
  TypeCode type;
  Any value;
};

interface TypedefDef : Contained,IDLType {};

struct TypeDescription {
  Identifier name;
  RepositoryId id;
  RepositoryId defined_in;
  VersionSpec version;
  TypeCode type;
};

interface StructDef : TypedefDef
{
  attribute StructMemberSeq members;
};

interface UnionDef : TypedefDef
{
  readonly attribute TypeCode discriminator_type;
  attribute IDLType discriminator_type_def;
  attribute UnionMemberSeq members;
};

interface EnumDef : TypedefDef
{
  attribute EnumMemberSeq members;
```

```
};
interface AliasDef : TypedefDef
{
  attribute IDLType original_type_def;
};
interface PrimitiveDef : IDLType
{
  readonly attribute PrimitiveKind kind;
};
interface StringDef : IDLType
{
  attribute unsigned long bound;
};
interface SequenceDef : IDLType
{
  attribute unsigned long bound;
  readonly attribute TypeCode element_type;
  attribute IDLType element_type_def;
};
interface ArrayDef : IDLType
{
  attribute unsigned long length;
  readonly attribute TypeCode element_type;
  attribute IDLType element_type_def;
};
interface ExceptionDef : Contained
{
  readonly attribute TypeCode type;
  attribute StructMemberSeq members;
};
struct ExceptionDescription {
  Identifier name;
  RepositoryId id;
  RepositoryId defined_in;
  VersionSpec version;
  TypeCode type;
};
enum AttributeMode {
  ATTR_NORMAL,
```

```
    ATTR_READONLY
  };

  interface AttributeDef : Contained
{
    readonly attribute TypeCode type;
    attribute IDLType type_def;
    attribute AttributeMode mode;
  };

  struct AttributeDescription {
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
    TypeCode type;
    AttributeMode mode;
  };

  enum OperationMode {
    OP_NORMAL,
    OP_ONEWAY
  };

  enum ParameterMode {
    PARAM_IN,
    PARAM_OUT,
    PARAM_INOUT
  };

  struct ParameterDescription {
    Identifier name;
    TypeCode type;
    IDLType type_def;
    ParameterMode mode;
  };
  typedef sequence <ParameterDescription> ParDescriptionSeq;

  typedef Identifier ContextIdentifier;
  typedef sequence <ContextIdentifier> ContextIdSeq;

  typedef sequence <ExceptionDef> ExceptionDefSeq;
  typedef sequence <ExceptionDescription> ExcDescriptionSeq;

  interface OperationDef : Contained
  {
    readonly attribute TypeCode result;
```

```
    attribute IDLType result_def;
    attribute ParDescriptionSeq params;
    attribute OperationMode mode;
    attribute ContextIdSeq contexts;
    attribute ExceptionDefSeq exceptions;
  };

  struct OperationDescription {
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
    TypeCode result;
    OperationMode mode;
    ContextIdSeq contexts;
    ParDescriptionSeq parameters;
    ExcDescriptionSeq exceptions;
  };

  typedef sequence <RepositoryId> RepositoryIdSeq;
  typedef sequence <OperationDescription> OpDescriptionSeq;
  typedef sequence <AttributeDescription> AttrDescriptionSeq;

  interface InterfaceDef : Container,Contained,IDLType
  {
    attribute InterfaceDefSeq base_interfaces;

    boolean is_a( in RepositoryId interface_id );

    struct FullInterfaceDescription
      Identifier name;
      RepositoryId id;
      RepositoryId defined_in;
      VersionSpec version;
      OpDescriptionSeq operations;
      AttrDescriptionSeq attributes;
      RepositoryIdSeq base_interfaces;
      TypeCode type;
    ;

    FullInterfaceDescription describe_interface();

    AttributeDef create_attribute(
      in RepositoryId id,
      in Identifier name,
      in VersionSpec version,
```

```
      in IDLType type,
      in AttributeMode mode );

    OperationDef create_operation(
      in RepositoryId id,
      in Identifier name,
      in VersionSpec version,
      in IDLType result,
      in OperationMode mode,
      in ParDescriptionSeq params,
      in ExceptionDefSeq exceptions,
      in ContextIdSeq contexts );
  };

  struct InterfaceDescription {
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
    RepositoryIdSeq base_interfaces;
  };
};
```

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison Wesley, 1986.

[2] Juan M. Andrade, Mark T. Carges, Terence J. Dwyer, and Stephen D. Felts. *The Tuxedo System.* Addison Wesley, 1996.

[3] Mark Betz. Interoperable objects. *Dr Dobbś Journal*, pages 18–39, October 1994.

[4] Grady Booch. *Object Oriented Design.* Benjamin Cummings, 1991.

[5] Open Software Foundation. *Introduction to OSF DCE.* Prentice Hall, 1992.

[6] Carl L. Hall. *Technical Foundations of Client/Server Systems.* Wiley, 1994.

[7] M. E. Lesk and E Schmidt. Lex - a lexical analyzer generator. *Computing Science Technical Report 39*, 1975.

[8] Object Management Group (OMG). *OMG Object Management Architecture Guide (OMA Guide), Revision 1.0.* Object Management Group (OMG), November 1990. OMG TC Document 90.9.1.

[9] Object Management Group (OMG). *The Common Object Request Broker: Architecture and Specification.* Object Management Group, 1993.

[10] Object Management Group (OMG). *IDL C++ Language Mapping Specification.* Object Management Group (OMG), September 1994. OMG TC Document 94.9.14.

[11] Object Management Group (OMG). *OMG RFP Submission – Interface Repository.* Object Management Group (OMG), November 1994. OMG TC Document 94.11.7.

[12] Object Management Group (OMG). *ORB 2.0 RFP Submission – ORB Initialisation Specification.* Object Management Group (OMG), September 1994. OMG TC Document 94.9.46.

[13] Object Management Group (OMG). *ORB 2.0 RFP Submission – ORB Interoperability.* Object Management Group (OMG), March 1994. OMG TC Document 94.3.1.

[14] Object Management Group (OMG). *ORB 2.0 RFP Submission – Universal Networked Objects.* Object Management Group (OMG), September 1994. OMG TC Document 94.9.32.

[15] Object Management Group (OMG). *CORBA 2.0 Interoperability Draft – Universal Networked Objects*. Object Management Group, March 1995. OMG TC Document 95.3.10.

[16] Robert Orfali and Dan Harkey. Clientserver with distributed objects. *Byte Magazine*, pages 151–162, April 1995.

[17] Ward Rosenberry, David Kenney, and Gerry Fisher. *Understanding DCE*. O'Reilly & Associates, Inc., 1992.

[18] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall International, 1991.

[19] John Shirley. *Guide to Writing DCE Applications*. O'Reilly & Associates, Inc., 1992.