New Jersey Institute of Technology

Digital Commons @ NJIT

Fall 1-31-1999

Comparison of DCOM and CORBA distributed computing

Reginald J. Reynolds
New Jersey Institute of Technology

Follow this and additional works at: https://digitalcommons.njit.edu/theses

Part of the Computer Sciences Commons

Recommended Citation

ABSTRACT

COMPARISON OF DCOM AND CORBA DISTRIBUTED COMPUTING

by
Reginald J. Reynolds

The movement of distributed applications from 2-tier to n-tier architectures have enabled systems to be scaled to meet the demands of an ever increasing population of users. Two middleware architectures have come to the forefront: Microsoft's DCOM and the OMG's CORBA. These are not the only possible architectures for n-tier distributed applications, but they are currently the only two which offer a degree of platform independence and the flexibility of using different programming languages for development.

CORBA provides platform independence because it provides a middle layer between the client and the server and services client requests using its internal naming service to identify server objects and then expose methods to the client through it's object adapter (POA). CORBA is a self contained middleware that operates independent of the underlying operating system. CORBA offers the potential of ease of maintainability since server objects can be changed and the new methods can be discovered at runtime by the client using CORBAs Dynamic Invocation Interface. Client code would therefore not have be recompiled as it would using static IDL mappings and client and server stubs.

DCOM, in contrast is a platform dependent solution that can only be used on Windows machines, although ports for other platforms are in the works. It relies on the Windows registry to identify objects and the operating system to assist in runtime control of objects. Because DCOM is nothing more of a remote extension to the already

established Common Object Model which all contemporary Windows operating systems and applications are built upon, it may provide the easiest path to distributed applications for Windows developers that are already familiar with the Common Object Model.

COMPARISON OF DCOM AND CORBA DISTRIBUTED COMPUTING

by
Reginald J. Reynolds

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
In Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Science

Department of Computer and Information Science

January 1999

Blank Page

APPROVAL PAGE

COMPARISON OF DCOM AND CORBA DISTRIBUTED COMPUTING

Reginald J. Reynolds

Dr. Vassilka Kirova, Thesis Advisor   Date
Visiting Professor of Computer Science, NJIT

Dr. Fadi P. Deek, Committee Member   Date
Vice Chairperson and Associate Professor of
Computer Science, NJIT

Dr. Phillip A. LaPlante, Committee Member   Date
President, Pennsylvania Institute of Technology

BIOGRAPHICAL SKETCH

Author: Reginald J. Reynolds

Degree: Master of Science

Date: January 1999

Undergraduate and Graduate Education:

- Master of Science in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 1999

- Master of Science in Administration,
  Central Michigan University, Mt. Pleasant, MI, 1995

- Bachelor of Science in Political Science,
  Tuskegee University, Tuskegee, AL, 1987

Major: Computer Science

To my beloved wife, whose unwavering support
made this accomplishment possible and my parents
who inspired me to strive for excellence through perseverance.

ACKNOWLEDGMENT

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Objective

The objective of this thesis is to present an architectural discussion of DCOM and

CORBA, the two contemporary architectures for development of n-tier applications, and

to compare the strengths and weaknesses of the two architectures when implemented on

computers running the Microsoft Windows operating system.

For the DCOM architecture, we examine how server objects are developed using

the *IUnknown* interface which allows multiple interfaces to be implemented for a single

server object and how these interfaces are registered in the Windows registry which acts

as a object repository for DCOM objects. We then discuss how server objects are

instantiated and controlled within the framework of the Microsoft Transaction Server, a

proprietary ORB that is integrated with the Windows operating system, runtime

environment.

For the CORBA architecture, we examine how server objects are developed

inheriting properties from *CORBA::Object* which allows server objects to use multiple

inheritance to expose different methods to different clients at runtime. Object

instantiation will be examined using both the traditional compiled client /server IDL

mappings and dynamic invocation of method using the Dynamic Invocation Interface

which allows clients to invoke methods at runtime that may not have been identified in

the client stub.

We will compare and contrast implementation issues using both the DCOM and CORBA architectures and discuss the strengths and weaknesses of both architectures when implemented on the Windows platform.

1.2 Background Information

The evolution of client server architecture has evolved in close parallel to that of the hardware platforms that it could be implemented on. Today client server models are categorized into three categories. We will refer to these models as thick client, thin client, and n-tier. We could just as easily apply the terminology of centralized computing, a topology that is reminiscent of X terminals being serviced by a mainframe, single-tier client server, the topology used in self contained local area networks prior to the mid nineties, or enterprise computing.

One might ask the question, why the paradigm shifts. The one answer lies in technology improvements that allowed hardware to be produced at lower cost driving the costs of ownership down. Today the average desktop computer has the computing power that a mainframe had twenty years ago. Dumb terminals of the past could only act as thin clients because of their lack of computing power. Today's desktops have the processing power and memory to act as both clients and servers. This was one of the driving forces of the paradigm shift to the thick client architecture where much of the processing requirements are shifted to the client allowing the server to provide services to a larger number of clients.

The other force was development of Windows based network operating systems such as Novell's Netware or Microsoft's NT. Unix, an advanced network operating

system, never really became widespread in the business and personal computer market because of it's complexity and it's original lashing to RISC type architectures. Recognizing, that the Unix operating system is over twenty years old, it was perhaps decades ahead of it's time. When Unix first introduced few outside of the technology industries, research organizations or academic institutions could afford the hardware to run it on. An added capability was that it had networking capabilities built into from the beginning. Both are excellent operating systems for the environments they were designed for. The domain of Unix is the high end technology, research and development, and academic environment. NT, the fastest growing Windows based operating system was designed to meet the need of the business environment. The scope of this document is; however, distributed application development on the Windows platform.

The development of the Intel 8086 architecture provided an affordable computing solution to business and home users, and the computer built upon this architecture were called personal computer (PCs). The operating system that was developed to run on this architecture was Microsoft's DOS (disk operating system). Unlike Unix, DOS was designed as a standalone operating system.

Client-server architectures fall into three categories referred to as thin client, thick client, and three tier (sometimes called n-tier). These names are based on whether the client or the server carry the burden of processing requirements. An example of a thin client architecture is a centralized system where a mainframe services requests from X terminals. The total processing burden lies on the server. The terminals can in fact be dumb meaning they have little or no processing capability of their own. As standalone computers gained processing power due to technological advances and subsequent

reduction in computer costs, thick client technologies were implemented and was the predominant model in the PC environment until the mid-nineties when PC based network operating systems such as Microsoft's Windows NT and Novell's Network Operating system came into vogue. In a thick client architecture, resources can be centralized, but once accessed by a client, the client carries the brunt of the processing requirements. Thick client architectures, for this reason, require a more robust client. Example of the thick client architecture include a client computer accessing a server based database for data, and then the data being processed on the client. Another example of the thick client architecture is the use of JAVA applets, which can be centrally located on a server and then distributed to clients through a web browser. Once the JAVA applets are downloaded by the client, the small programs are compiled and executed on the client. Three-tier or n-tier architecture applies to a system where multiple servers are used to service requests by clients. In an n-tier environment, resources can be centrally located and processing tasks can be distributed between the client and multiple server machines. In an n-tier architecture, a single server can service multiple clients and in some instances simultaneously if an object resource broker is used to manage concurrency and atomicity of transactions. The ability to distribute processing requirements along while maintaining concurrency of transactions gives the n-tier model the ability to have unlimited scalability potential. An example of an implementation of an n-tier architecture would be a web based system where the client requests information via an application that runs in a web browser, the request is serviced by a web server that is collocated with an object resource broker that contains objects that can be invoked to return data from an database server such as Oracle or SQL server located on a remote computer. The object resource broker

controls the transaction to include opening and closing a database connection on a remote computer and returns the information to the client application running in the client's web browser.

CHAPTER 2

PROBLEM DESCRIPTION AND METHODOLOGY

2.1 Problem Domain

This thesis will provide a comparison and contrast of the two most prominent distributed architectures on the Windows platform.

2.2 Problem Statement

Examine both the DCOM and CORBA architectures and support the hypothesis that DCOM is a better architecture for distributed application development on Windows platforms.

2.3 Conceptual Development

Conceptual development will begin with a discussion of current distributed applications paradigms and the capabilities and obstacles they face in the contemporary network environment. The analysis will start on a macro scope with a discussion of Common Object Resource Broker Architecture (CORBA) which the Object Modeling Group (OMG) maintains proponency for and Microsoft's Distributed Object Model (DCOM). The focus will be on the development issues of server objects using each of the architectures above since both DCOM and CORBA based applications can service client applications which reside on any platform, Windows, Unix, Macintosh etc.

2.4 Sources

We will use secondary sources such as books, magazine articles, white papers, and case studies to provide the technical basis of my research. Sources will include both hard copy

and internet media. Assumptions will be based on observations from the application

development of others and my own personal experience using the two architectures.

2.5 Glossary

Client – A process that invokes a server object's methods.

Interface – A named collection of abstract operations or methods that represent one functionality.

Object class (or class) – A named concrete implementation of one or more interfaces.

Object (or object instance) – An instantiation of some object class.

Object server – A process responsible for creating and hosting object instances.

Server object – A process responsible for providing services for a client.

CHAPTER 3

ARCHITECTURAL DISCUSSION

3.1 Architectural Discussion of DCOM

Understanding DCOM requires that, you understand COM since DCOM is nothing more

than the Common Object Model architecture extended across the network using DCE

with an underlying wire protocol such as TCP/IP. COM is a binary standard for objects.

It defines how an object should identify itself to the system after it has been compiled

using a target language into machine code. Any programming language can be used to

develop COM objects as long as it can implement COM compliant interfaces. There are

current COM mappings for C++, Java, and Visual Basic. Scripting languages such as

JScript and VbScript can be used to implement COM interfaces. Objects written in

different languages can communicate to each other because they recognize and can

implement the methods exposed by the COM interfaces (Jennings 1997).

COM evolved from Object Linking and Embedding (OLE). OLE allows

applications to exchange and display information without knowing anything about how

the methods in the sharing application are implemented. All recent Windows operating

systems, Windows 95, Windows 98, and Windows NT 4.0 are built upon COM. COM

objects are identified by the operating system via two mechanisms: class IDs (CLSIDs)

and program IDs (ProgIDs). These two identification mechanisms are stored in the

Windows registry (Jennings 1997). At this point in the discussion of DCOM and

CORBA, I would like to point out a primary difference in the approach of the two

architectures. DCOM a subset of COM is tightly interwoven with the Windows operating

system thus DCOM servers must be developed and implemented on a Windows platform.

The CORBA architecture is self contained and independent of the underlying operating system thus CORBA servers can be developed and implemented on any platform. It should be noted; however, that implementations of DCOM are currently being developed for Unix that provide a Windows registry on top of the Unix operating system. SAG provides one such implementation and is freely downloadable in beta format from the internet.

The Distributed Common Object Model (DCOM) refers to a complete distributed computing framework which includes a distributed computing architecture, but a proposed protocol standard as well. The DCOM protocol, also known as Object RPC (ORPC), is a set of definitions that extend the standard DCE RPC protocol. We will use DCOM and ORPC interchangeably throughout my discussion of the DCOM architecture. It has been designed specifically for the DCOM object oriented environment, and specifies how calls are made across a network and how references to objects are represented and maintained. The ORPC protocol has been submitted as an Internet Draft to the Engineering Task Force (IETF) by developers at Microsoft Corporation, as it is suited to both Internet and Intranet component communication. The status of this document is a "work in progress" and will be referenced throughout this document as (Brown 98). DCOM is built upon two already established computing architectures. The first is DCE RPC, which is an established internet standard for remote procedure calls. The second is Microsoft's Common Object Model (COM) which is the framework that the Windows NT operating system is built upon. Since this document focuses on the DCOM and CORBA architectures, we will only discuss underlying architectures to the

extent necessary to understand the DCOM or CORBA architectures. The figure below

taken from (Brown 98), depicts a typical DCOM architecture.
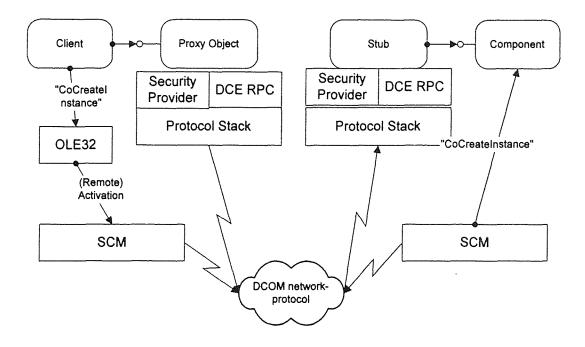
Figure 3.1: DCOM Architecture

At the wire level (network interface), ORPC uses standard RPC packets, with

additional DCOM-specific information—in the form of an Interface Pointer Identifier

(IPID), version information, and extensibility information—conveyed as additional

parameters on calls and replies. The IPID is used to identify a specific interface on a

specific object on a server machine where the procedure call will be processed. The

marshaled data on an ORPC packet is stored in standard Network Data Representation

(NDR) format, so issues of byte order or floating point formats are automatically handled.

DCOM uses one new NDR type, which represents a marshaled interface. DCOM client

machines are responsible for periodically ensuring that objects are kept alive on the server

by background pinging. This process has been optimized to reduce unnecessary pinging

and minimize network traffic. One thing that should be noted about the pings that DCOM

relies upon to maintain the lifetime of instantiated server objects, is that it is interface

based instead of object based. This makes since because DCOM implementations use

inheritance to expose multiple interfaces of objects. All interfaces that an object

possesses may not be utilized; however, to perform a given transaction. A technology

called "Delta Pinging" combines all necessary interfaces in a given object or objects to be

combined in a set. The set can then be pinged in the background reducing network traffic

because each interface doesn't have to be pinged individually (Brown 98).

Programmers for the most part, do not have to work at the ORPC level. The

Microsoft IDL compiler (MIDL) can be used to automatically generate the code that is

needed to transfer the data across the network, based simply on an IDL file (Brown 98).

3.2 Architectural Discussion of CORBA

The Common Request Broker Architecture (CORBA), is the Object Management

Group's (OMG) answer to the need for a solution that provide interoperability between

the rapidly proliferating number of hardware and software products available today.

Simply stated, CORBA allows applications to communicate with one another regardless

of where they are located or what hardware or software platform they reside on. CORBA

1.1 was introduced in 1991 by the OMG and defined interface definition language (IDL)

and the Application Programming Interfaces (API) that enable client-server object

interaction within a specific implementation of an Object Resource Broker (ORB). The

CORBA 2.0 specification, adopted in 1994, defined true interoperability by specifying

how ORBs from different vendors can interoperate (OMG 1998).

The following figure shows the primary components in the OMG Reference

Model architecture. Definitions are provided below. Portions of these descriptions, such

as the figure below, are based on material from Vinoski.
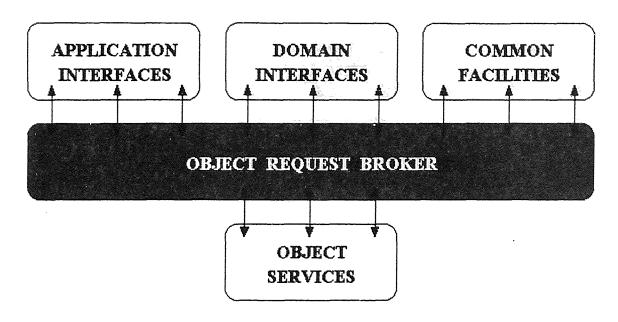
Figure 3.2: OMG Reference Model Architecture

- **Object Services** - - Domain-independent interfaces that are used by many distributed
  object programs. Two examples of Object Services that fulfill this role are:

  - The Naming Service - - which allows clients to find objects based on names;
  - The Trading Service - - which allows clients to find objects based on their
    properties.
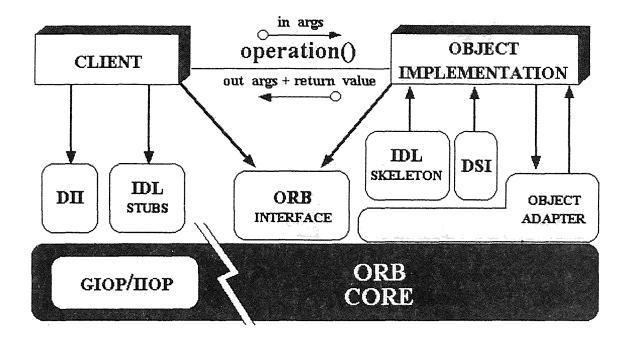
There are also Object Services specifications for lifecycle management, security,
transactions, and event notification, as well as many others (OMG 95)
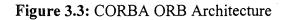
- **Common Facilities** - - These interfaces are also horizontally-oriented, but unlike
  Object Services they are oriented towards end-user applications. An example of such
  a facility is the *Distributed Document Component Facility* (DDCF), a compound
  document Common Facility based on OpenDoc. DDCF allows for the presentation
  and interchange of objects based on a document model, an example would be the
  linking of a spreadsheet object into a report document.

- **Domain Interfaces** - - These interfaces fill roles similar to Object Services and Common Facilities but are oriented towards specific application domains. One example are the OMG RFPs for the manufacturing domain. Other OMG RFPs have been issued for the telecommunications, medical, and financial domains.

- **Application Interfaces** - - These are interfaces developed specifically for a given application. Because they are application-specific, and because the OMG does not develop applications (only specifications), these interfaces are not standardized. However, if over time it appears that certain broadly useful services emerge out of a particular application domain, they might become candidates for future OMG standardization (Schmidt 1998).

The ORB is the middleware that establishes the client-server relationships between objects. The ORB acts as a middle man allowing a client to transparently invoke a method on a server object, which can reside locally on the same machine or across a network.. The ORB intercepts the call and is responsible for finding an object that can implement the request, marshal the parameters, send the parameters, unmarshal the parameters on the other end, and return the results. The client does not have to be aware of the server object's location, what language it was programmed in, or the underlying operating system on the server machine (OMG 1998).

The figure below, taken from the OMG CORBA specification, illustrates the primary components in the CORBA ORB architecture. Descriptions will follow below the figure on the next page.

Figure 3.3: CORBA ORB Architecture

- **Object Implementation - -** This defines operations that implement the CORBA IDL interface. Object implementations can be written in a variety of programming languages. There are current CORBA mappings for C, C++, Java, Smalltalk, and Ada.

- **Client - -** This is the application that requests services from a server object. Accessing the services from a remote object should be transparent to the requesting application.

- **Object Request Broker (ORB) - -** The ORB, simply put is the object bus. The ORB provides a mechanism which facilitates transparent communication between client applications and server objects which contain methods and their implementations. The ORB simplifies distributed programming by decoupling the client from the details of the method invocations. When a client invokes an operation, the ORB is responsible for finding the server object implementation, transparently activating it if necessary, delivering the request to the server object, and returning values to the client application.

- **ORB Interface - -** An ORB is a logical entity that may be implemented in various ways (such as one or more processes or a set of libraries). To decouple applications from implementation details, the CORBA specification defines an abstract interface for an ORB. This interface provides various helper functions as converting object

references to strings and vice versa, and creating argument lists for requests made through the dynamic invocation interface described below.

- **CORBA IDL Stubs and Skeletons** - - CORBA IDL stubs (client IDL) and skeletons (server IDL) serve as the "glue" between the client and server applications, respectively, and the ORB. The transformation between CORBA IDL definitions and the target programming language is automated by a CORBA IDL compiler. The use of an IDL compiler reduces the potential for inconsistencies between client stubs and server skeletons and increases opportunities for automated compiler optimizations.

- **Dynamic Invocation Interface (DII)** - - This interface allows a client to directly access the underlying request mechanisms provided by an ORB. Applications use the DII to dynamically issue requests to objects without requiring IDL interface-specific stubs to be linked in. Unlike IDL stubs (which only allow RPC-style requests), the DII also allows clients to make non-blocking *deferred synchronous* (separate send and receive operations) and *oneway* (send-only) calls.

- **Dynamic Skeleton Interface (DSI)** - - This is the server side peer to the client DII. The DSI allows an ORB to deliver requests to an object implementation that does not have compile-time knowledge of the type of the type of the object it is implementing. The client making the request has no idea whether the implementation is using the type-specific IDL skeletons or is using dynamic skeletons.

- **Object Adapter** - - This assists the ORB with delivering requests to the object and with activating the object. More importantly, an object adapter associates object implementations with the ORB. Object adapters can be specialized to provide support for certain implementation styles (such as OODB object adapters for persistence and library object adapters for non-remote objects) (Schmidt 1998).

The basic functionality provided by the ORB consists of passing requests from clients to the server objects where the implementations are invoked. In order for a client to make a request the client can communicate with the ORB Core either through the IDL stub or through the DII. The client stub represents the mapping between the language the client application was written in and the ORB core. The ORB core then transfers the request to the server object which receives the request and implements it (Kksiazek 1998).

The communication between the server object implementation and the ORB core is handled by the Object Adapter. OMG specifies four policies in which the OA may

handle server object implementation. These are *shared server policy, unshared server policy, server-per-method policy*, and *persistent server policy*. Using shared server policy, multiple objects may be activated by the same program. Unshared server policy is used to activate server objects individually. Server-per-method policy policy occurs when a new instance of the server object is created each time a request is received. When persistent server policy is used the server object must be constantly active. In order for the OA to implement a given policy, it must have access to information about the location of server objects and their run-time environments. This information is maintained in the *Implementation Repository* which is a standard component of the CORBA architecture (Kksiazek 1998).

The interfaces of server objects can be specified in two ways, OMG IDL, or by adding them to the Interface Repository. The DII allows the client to specify requests to server objects whose definition whose definition and interface are unknown at the client's compile time. On the server side the DSI allows the skeletons to be bypasses and methods to be invoked dynamically. This allows programmers to redesign server objects without requiring new client and server stubs to be generated (Kksiazek 1998).

Implementational differences are not the only obstacles that separate objects. Other barriers might include security requirements or requirements placed on the development environment. In order to provide interoperability, the higher-level domain concept was introduced in the CORBA 2.0 specification. This concept segregates server objects which for some reason, be it implementational or administrative, must be separated from other objects. A bridging mechanism (mapping between domains) is required to interact with these objects (Kksiazek 1998).
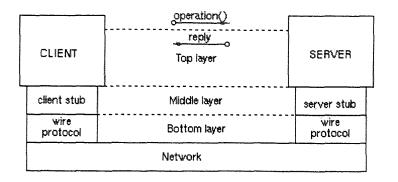
The interoperability approaches are generally divided into immediate and mediated bridging. Mediated bridging occurs when the interacting elements of one domain are transformed at the boundary of each domain. This common form could be either standard (specified by OMG, for example IIOP), or a private agreement between the two parties. Immediate bridging occurs when elements of one domain are transformed directly between the internal form of one domain and the other. Furthermore if the mediation is internal to one execution environment (for example TCP/IP) it is known as a "full bridge", otherwise if the execution environment of one ORB is different from the common protocol, then each ORB can be called a "half bridge" (Kksiazek 1998).

In order to make bridges possible, it was necessary for a standard transfer protocol. This function is fulfilled by the General Inter-ORB Protocol (GIOP). Defined by the OMG, it has been specifically defined to meet the needs of ORB to ORB interaction and is designed to work over any transport that meets a minimal set of assumptions. Apart from defining the general transfer syntax, the OMG also specified how it is going to be implemented using the TCP/IP transport and called it Internet Inter-ORB Protocol (IIOP). The OMG points out that the relationship between GIOP and IIOP is the same as between IDL and its concrete mapping, for example a C++ mapping. IIOP is designed to provide "out of the box" interoperability with other compatible ORBs since TCP/IP is the most popular vendor-independent transport layer (Kksiazek 1998).

3.3 Architectural Comparison of CORBA and DCOM

CORBA and DCOM are both viable frameworks from which distributed client-server systems can be developed on Windows platforms. The description about DCOM is based
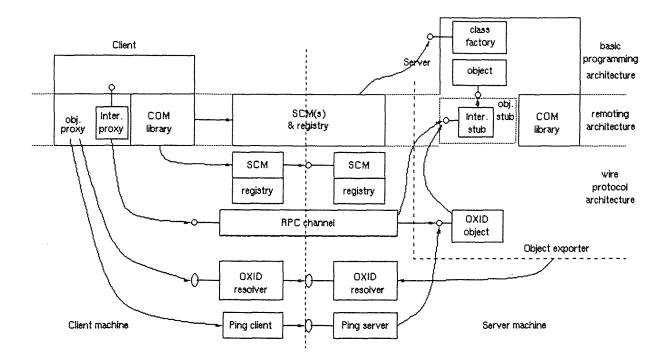
on the COM specification (COM 95) and the DCOM specification (Brown 98). The CORBA description is based on the CORBA specification outlined by the OMG (CORBA 95). When a client requests a service, it invokes a method implemented by a remote object, which acts as the server in the client-server model. Both CORBA and DCOM rely on an object request broker (ORB) to instantiates the server object and control its' process on the server platform. CORBA has numerous implementations from many vendors that each have their own proprietary ORB. Microsoft's DCOM relies on the Microsoft Transaction Server to control server objects on Windows NT platforms. The interface of the server object in both implementations is described in IDL. Each CORBA vendor has it's own proprietary IDL compiler. Microsoft's version is called the MIDL compiler. These compilers read IDL files, conduct the appropriate mappings based on the target language the server is written in, and create the necessary client and server stubs which are used for communication between the client and the server. Methods are encapsulated and exposed in the server stub which can be used by developers of client front ends. The actual implementation of these methods is hidden from the client. Object oriented programming features are present at the IDL level, such as data encapsulation, polymorphism and inheritance. CORBA and DCOM support different aspects of these features; however. CORBA supports multiple inheritance and exception handling at the IDL level. DCOM instead supports multiple interfaces to achieve a similar purpose. Current implementations of DCOM support exception handling as well (Chung 1997). The figure on the following page is from (Chung 1997).

Figure 3.4: RPC structure

The interactions between a client process and an object server are implemented as object-oriented RPC communications in both CORBA and DCOM (Birrell 84). The Figure below shows a typical RPC structure. To invoke a remote function, the client makes a call to the client stub. The stub packs the call parameters into a request message, and invokes a wire protocol to send the message to the server. On the server side, the wire protocol delivers the message to the server stub, which in turn unpacks the request message and invokes the method on the server object. In CORBA, the client stub is referred to as the *stub* and the server stub the *skeleton*. DCOM, in contrast, the client stub is called the *proxy* and the server stub is called the *stub* (Chung 1997).

The overall architectures of CORBA and DCOM can be sub-divided into three layers. We will refer to these layers as the *basic programming architecture, the remote architecture,* and the *wire protocol architecture.* The basic programming architecture is the top layer and is visible to developers of client and object server programs. The remote architecture is the middle layer and makes interface pointers or object references meaningful across processes. The wire protocol architecture is the bottom layer which is used to extend the remote architecture to work across different machines on a network. The remaining figures and tables in this section are taken from (Chung 1997).

The two architectures are depicted in the figures below.

Figure 3.5: DCOM Architecture
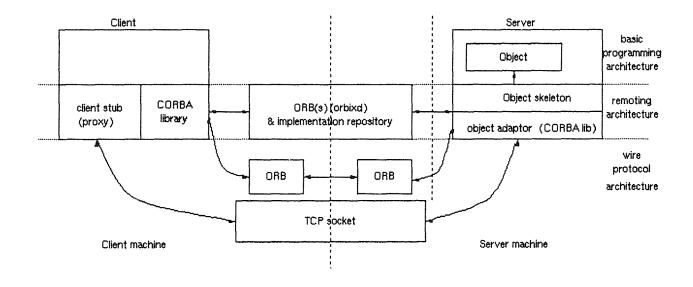
Figure 3.6: CORBA Architecture

We will use code "snippets" from a sample application to exemplify some of the differences between CORBA and DCOM implementations. CORBA and DCOM IDL code is different due the way inheritance is implemented in the two frameworks. Remember that DCOM support objects with multiple interfaces, while CORBA allows a single interface to inherit from multiple interfaces. These differences will manifest themselves when you examine how methods are exposed to the client. For this reason developers of client applications will have to use two different approaches for client development, one for use with DCOM servers and one for CORBA servers. Examine the IDL code in Table 1.

The CORBA IDL defines three interfaces: (1) interface *grid1* supports the *get()* and *set()* methods; (2) interface *grid2* supports the *reset()* method; (3) interface *grid* multiply inherits from *grid1* and *grid2*. DCOM, in contrast defines two interfaces, *Igrid1* and *Igrid2*, for the two groups of methods. The implementation of the *Grid* object uses multiple inheritance from *Igrid1* and *Igrid2* to implement a server object with two interfaces (Chung 1997).

Compiling the IDL files with an IDL compiler generates the proxy/stub/skeleton code and the interface header file *grid.h* or *grid.hh* that are used by both the client and the server. One should note that in DCOM, each interface has a *globally unique identifier* (GUID) referred to as the *interface ID* (IID). Each object class in the DCOM IDL is similarly assigned a unique *class ID* (CLSID). This is due to the tight integration of DCOM with the Windows NT operating system, which is built around the *common object model* (COM). DCOM in fact is nothing more than COM distributed across multiple machines. Every DCOM interface must inherit from the *IUnknown*

Table 3.1: The IDL Files

DCOM IDL	CORBA IDL

`// uuid and definition of IGrid1`
`[ object,`
`    uuid(3CFDB283-CCC5-11D0-BA0B-00A0C90DF8BC),`
`    helpstring("IGrid1 Interface"),`
`    pointer_default(unique)`
`]`
`interface IGrid1 : IUnknown    {`
`    import "unknwn.idl";`
`    HRESULT get``([in] SHORT n, [in] SHORT m, [out] LONG *value);`
`    HRESULT set``([in] SHORT n, [in] SHORT m, [in] LONG value);`
`    };`
	`interface grid1`
	`{`
	`    long `**`get`**`(in short n, in short m);`
	`    void `**`set`**`(in short n, in short m, in long value);`
	`};`

`// uuid and definition of IGrid2`
`[ object,`
`    uuid(3CFDB284-CCC5-11D0-BA0B-00A0C90DF8BC),`
`    helpstring("IGrid2 Interface"),`
`    pointer_default(unique)`
`]`
`interface IGrid2 : IUnknown    {`
`    import "unknwn.idl";`
`    HRESULT reset``([in] LONG value);`
`    };`
	`interface grid2`
	`{`
	`    void `**`reset`**`(in long value);`
	`};`

`// uuid and definition of type library`
`[  uuid(3CFDB281-CCC5-11D0-BA0B-00A0C90DF8BC),`
`    version(1.0),`
`    helpstring("grid 1.0 Type Library)`
`]`
`library GRIDLib`
`{`
`    importlib("stdole32.tlb");`
`    // uuid and definition of class`
`    [  uuid(3CFDB287-CCC5-11D0-BA0B-00A0C90DF8BC),`
`        helpstring("Grid Class")`
`    ]`
`    // multiple interfaces`
`    coclass CGrid`
`    {  [default] `**`interface IGrid1`**`;`
`        interface IGrid2``;`
`    };`
`};`
	`// multiple inheritance of interfaces`
	`interface grid: grid1, grid2`
	`{`
	`};`

interface which provides a *QueryInterface()* method for navigating between interfaces in

the same object, and the *AddRef()* and *Release()* methods used for reference counting.

Reference counting is a mechanism that allows a COM object to keep track of its clients

and allows it to delete itself when it is no longer needed (Chung 1997).

In the CORBA implementation, the application developer writes the implementation class *grid_i*. There are two approaches to associating the implementation class with the interface class, the inheritance approach and the delegate approach. The inheritance approach is the one that is used in the sample application. Using this approach, the Orbix IDL compiler, which was used to compile the IDL files in CORBA example, also generates a class called *gridBOAImpl* that is responsible for instantiating the skeleton (server stub) class. Class *gridBOAImpl* inherits from the interface class *grid*, which inherits from class *CORBA::Object*. The implementation class *grid_i* inherits from class *gridBOAImpl* to complete the mapping between the interface class and the implementation class. It should be noted that class *gridBOAImpl* is proprietary to the Iona Orbix version of CORBA, since the original CORBA specification by the OMG does not specify what the skeleton (server stub) class should look like or what the name of the base class should be. To resolve this issue, the *Portable Object Adapter* (POA) was introduced. (POA 97) The incorporation of the POA specifies the name of the base class of the server stub (Chung 1997).

The tables 3.2 and 3.3 on the following pages show header and implementation files for the server class implemented in DCOM and CORBA. Examining the DCOM program, an event is created and waits on that event, which is signaled when all active server objects are deleted so the server can exit. The actual client requests are handled concurrently by different threads from a thread pool (Chung 1997).

Table 3.2: The server implementation header files

DCOM server class definition (cgrid.h)	CORBA server class definition (grid_i.h)
| ```
#include "grid.h" // IDL-generated interface header file

class CClassFactory : public IClassFactory {
public:
// IUnknown
STDMETHODIMP QueryInterface(REFIID riid, void** ppv);
STDMETHODIMP_(ULONG) AddRef(void) { return 1; };
STDMETHODIMP_(ULONG) Release(void) { return 1; }

// IClassFactory
STDMETHODIMP CreateInstance(LPUNKNOWN punkOuter,
            REFIID iid, void **ppv);
STDMETHODIMP LockServer(BOOL fLock)
      { return E_FAIL; };
};

class CGrid : public IGrid1, public IGrid2 {
public:
// IUnknown
STDMETHODIMP QueryInterface(REFIID riid, void** ppv);
STDMETHODIMP_(ULONG) AddRef(void)
    { return InterlockedIncrement(&m_cRef); }
STDMETHODIMP_(ULONG) Release(void)
    { if (InterlockedDecrement(&m_cRef) == 0)
        { delete this; return 0; }
      return 1; }
// IGrid1
STDMETHODIMP get(IN SHORT n, IN SHORT m,
          OUT LONG *value);
STDMETHODIMP set(IN SHORT n, IN SHORT m,
          IN LONG value);
// IGrid2
STDMETHODIMP reset(IN LONG value);

CGrid(SHORT h, SHORT w);
~CGrid();
private:
LONG  m_cRef, **m_a;
SHORT m_height, m_width;
};
``` | ```
#include "grid.hh" // IDL-generated interface header file

class grid_i : public gridBOAImpl {
public:

virtual CORBA::Long get(CORBA::Short n, CORBA::Short m,
        CORBA::Environment &env);
virtual void set(CORBA::Short n, CORBA::Short m,
        CORBA::Long value, CORBA::Environment &env);

virtual void reset(CORBA::Long value,
          CORBA::Environment &env);
grid_i(CORBA::Short h, CORBA::Short w);
virtual ~grid_i();
private:
CORBA::Long **m_a;
CORBA::Short m_height, m_width;
};
``` |

Table 3.3: The server implementation files

DCOM server implementation	CORBA server implementation
| ```
#include "cgrid.h"

STDMETHODIMP
CClassFactory::QueryInterface(REFIID riid, void** ppv)
{
  if (riid == IID_IClassFactory || riid == IID_IUnknown) {
    *ppv = (IClassFactory *) this;
    AddRef(); return S_OK;
  }
  *ppv = NULL;
  return E_NOINTERFACE;
}

STDMETHODIMP
CClassFactory::CreateInstance(LPUNKNOWN p,
REFIID riid, void** ppv) {
  IGrid1* punk = (IGrid1*) new CGrid(100, 100);
  HRESULT hr = punk->QueryInterface(riid, ppv);
  punk->Release();
  return hr;
}

STDMETHODIMP CGrid::QueryInterface(REFIID riid,
void** ppv) {
  if (riid == IID_IUnknown || riid == IID_IGrid1)
    *ppv = (IGrid1*) this;
  else if (riid == IID_IGrid2) *ppv = (IGrid2*) this;
  else { *ppv = NULL; return E_NOINTERFACE; }
  AddRef(); return S_OK;
}

STDMETHODIMP CGrid::get(IN SHORT n, IN SHORT
m, OUT LONG* value) {
  *value = m_a[n][m];
  return S_OK;
}

STDMETHODIMP CGrid::set(IN SHORT n, IN SHORT
m, IN LONG value) {
  m_a[n][m] = value;
  return S_OK;
}
``` | ```
#include "grid_i.h"
















CORBA::Long grid_i::get(CORBA::Short n,
CORBA::Short m,
            CORBA::Environment &) {

  return m_a[n][m];
}

void grid_i::set(CORBA::Short n, CORBA::Short m,
    CORBA::Long value, CORBA::Environment &) {
    m_a[n][m] = value;
}

void grid_i::reset(CORBA::Long value,
CORBA::Environment &) {
    short n, m;
    for (n = 0; n < m_height; n++)
     for (m = 0; m < m_width; m++)
      m_a[n][m]=value;
    return;
}
grid_i::grid_i(CORBA::Short h, CORBA::Short w) {
    m_height=h;  // set up height
``` |

Table 3.3: The server implementation files (Continued)

```
STDMETHODIMP CGrid::reset(IN LONG value) {
    SHORT n, m;
    for (n=0; n < m_height; n++)
        for (m=0; m < m_width; m++)
            m_a[n][m] = value;
    return S_OK;
}


CGrid::CGrid(SHORT h, SHORT w) {
    m_height = h;
    m_width= w;
    m_a = new LONG*[m_height];
    for (int i=0; i < m_height; i++)
        m_a[i] = new LONG[m_width];
    m_cRef = 1;
}

extern HANDLE hevtDone;

CGrid::~CGrid () {
    for (int i=0; i < m_height; i++)
        delete[] m_a[i];
    delete[] m_a;
    SetEvent(hevtDone);
}
```

```
m_width=w;    // set up width
m_a = new CORBA::Long* [h];
    for (int i = 0; i < h; i++ )
        m_a[i] = new CORBA::Long[w];
}




grid_i::~grid_i () {
    for (int i = 0; i < m_height; i++)
        delete[] m_a[i];
    delete[] m_a;
}
```

The CORBA server program instantiates an instance of class *grid_i* and then

blocks at *impl_is_ready()* to receive incoming client requests. If the server does not

receive any requests before the default timeout period is reached (which can be set by the

programmer), it gracefully shuts down. Client requests can be handled either serially or

by different threads, depending on the activation policy used by the server object (Chung

1997).

DCOM client code tends to be longer than CORBA client code due to the

additional method calls to *IUnknown*. This may not be a true statement for DCOM clients

written in Visual Basic or Java, where the virtual machine layer takes care of the

IUnknown method calls relieving programmers of that responsibility (Chappell 97). Even

in a C++ client, smart interface pointers can be used to hide reference counting (Rogerson 96).

Both DCOM and CORBA, require a registration process of the server, prior to execution. In CORBA, the association between the interface name and the path name of the server executable is registered in the *implementation repository*. In DCOM, the association between the CLSID and the path name of the server executable is registered in the *registry*. In addition, since a DCOM interface proxy/stub is itself a COM object, it's associated in-process server (a dll that runs in the client process) must be registered. (note) The simplest method to accomplish this is to install the the dll as a component in Microsoft Transaction Server (MTS), the ORB designed for DCOM which has been incorporated into Microsoft Internet Information Server 4.0 which is part of the Windows NT Operating system.

DCOM utilizes a type library generated by the IDL compiler which, assigns a GUID, to store information for interface methods. It can be used through the *Idispatch* interface to perform dynamic invocation. (Rogerson 96) It can also be used for type library-driven marshaling: (Grimes 97) instead of using a separate proxy/stub DLL that contains information specific to an interface, a generic marshaler can perform marshaling by reading type library information. In CORBA, the IDL compiler generates the type information for each method in an interface and stores it in the *Interface Repository* (IR). A client can query the interface repository to get run-time information about a particular interface and then use that information to create and invoke a method on the object dynamically through the *dynamic invocation interface* (DII). Similarly, on the server side, the *dynamic skeleton interface* (DSI) allows a client to invoke an operation on an

object that has no compile time knowledge of the type of object it is implementing

(CORBA 95). Tables 3.4 and 3.5 are the main program files for the DCOM and CORBA

applications.

Table 3.4: The server main programs

DCOM server main program	CORBA server main program
``HANDLE hevtDone;

void main()
{
    // Event used to signal this main thread
    hevtDone = CreateEvent(NULL, FALSE, FALSE,
NULL);
    hr = CoInitializeEx(NULL,
COINIT_MULTITHREADED);
    CClassFactory* pcf = new CClassFactory;
    hr = CoRegisterClassObject(CLSID_CGrid, pcf,
        CLSCTX_SERVER, REGCLS_MULTIPLEUSE,
&dwRegister);
    // Wait until the event is set by CGrid::~CGrid()
    WaitForSingleObject(hevtDone, INFINITE);
    CloseHandle(hevtDone);
    CoUninitialize();
}``` | ```int main()
{
    // create a grid object using the implementation class grid_i

    grid_i ourGrid(100,100);

    try {
    // tell Orbix that we have completed the server's initialization:
    CORBA::Orbix.impl_is_ready("grid");
    } catch (...) {
    cout << "Unexpected exception" << endl;
    exit(1);
    }
}``` |

Table 3.5: The client main programs.

DCOM Client code	CORBA Client code
``#include "grid.h"

void main(int argc, char**argv)
{
    IGrid1   *pIGrid1;
    IGrid2   *pIGrid2;
    LONG   value;

    CoInitialize(NULL);   // initialize COM
    CoCreateInstance(CLSID_CGrid, NULL, CLSCTX_SERVER,
        IID_IGrid1, (void**) &pIGrid1);
    pIGrid1->get(0, 0, &value);
    pIGrid1->QueryInterface(IID_IGrid2, (void**) &pIGrid2);
    pIGrid1->Release();
    pIGrid2->reset(value+1);
    pIGrid2->Release();
    CoUninitialize();
}``` | ```#include "grid.hh"

void main (int argc, char **argv)
{
    grid_var gridVar;

    CORBA::Long value;

    // bind to "grid" object; Orbix-specific
    gridVar = grid::_bind(":grid");

    value = gridVar->get(0, 0);

    gridVar->reset(value+1);
}``` |

The three frameworks discussed earlier, *the basic programming architecture, the remote architecture*, and *the wire protocol architecture*, will be elaborated on as it applies to DCOM and CORBA. The basic programming architecture shows the programmers' view of DCOM and CORBA. More specifically, I will describe how a client requests an object and invokes its methods, and how a server creates an object instance and makes it available to the client. Exactly how the client is connected to server objects is totally hidden from the programmer. The client and server programs interact as if they reside in the same address space on the same machine. The main differences between DCOM and CORBA at this layer include how interfaces are specified by clients, and DCOM's class factories and *IUknown* methods. A step-by-step description is provided in Table 3.6 and broken down in figures 3.7 and 3.8 for DCOM and CORBA, respectively (Chung 1997).

Table 3.6 gives a common DCOM invocation sequence, there are a few things that should be pointed out. First, the use of class factories in COM is optional. A server object can actually call *CoRegisterClassObject()* to register any interface pointer, and clients can invoke another COM API named *CoGetClassObject()* to retrieve that pointer. (A class object is a container or metaclass for a COM object class) Second, *CoCreateInstance()* does not necessarily create a new instance. Inside *IClassFactory::CreateInstance()*, a server can choose to always return the same interface pointer so that different clients can connect to the same object instance at a particular state. Another method of binding to a specified server object is to use *monikers* (Box 2 97) and/or the *Running Object Table* (ROT) (COM 95).

An object can be activated by invoking any method on an existing object reference in CORBA. Some CORBA vendors provide special method calls, the *bind() operation in Orbix, to activate a server object and obtain its object reference. The client may attach to an existing instance instead of a new instance, if there is an existing instance matching the requested type. A client can store an object reference by stringifying it using object_to_string()* and can later use it again by converting it back to its original form by calling *string_to_object()* (Chung 1997).

Another difference between DCOM and CORBA at the programming layer is the way that exceptions are handled. CORBA provides support for standard C++ exceptions and some CORBA specific exceptions. In addition, user defined exceptions are also allowed and are declared in the IDL. The IDL compiler maps user defined exceptions to a C++ class. DCOM, in contrast, requires that all methods return a 32-bit error code called an *HRESULT* at this layer. (see Table 3.3) At the compiler level, a set of conventions and system provided services (called the *IErrorInfo* object) allows failure *HRESULTs* to be converted into exceptions in a way natural to the language being used. Programmers using Microsoft Visual C++ 5.0 or later, for example, can use standard C++ try/catch blocks to catch errors from COM method invocations. Similarly, some compilers allow programmers to "throw exceptions" instead of returning failure codes. The DCOM wire protocol includes a mechanism known as *body extensions* (Brown 96) that allow rich exception information (such as a string explaining the error) to be carried.

The middle layer (the remote architecture) consists of the necessary infrastructure to create the appearance that the client and server have the same address space. Another term for this is location transparency. The description in Table 3.7 shows how the server

Table 3.6: The top layer description.

DCOM	CORBA
Object activation	
1. Client calls COM library's `CoCreateInstance()` with `CLSID_Grid` and `IID_IGrid1`. 2. COM infrastructure starts an object server for `CLSID_Grid`. 3. As shown in the server main program, server creates class factories for all supported CLSIDs, and calls `CoRegisterClassObject()` to register each factory. Server blocks on waiting for, for example, an event to be set to signal that the server is no longer needed. Incoming client requests will be served by other threads. 4. COM obtains the `IClassFactory` pointer to the `CLSID_Grid` factory, and invokes `CreateInstance()` on it. 5. In `CreateInstance()`, server creates an object instance and makes a `QueryInterface()` call to obtain an interface pointer to the `IID_IGrid1` interface. 6. COM returns the interface pointer as `pIGrid1` to the client.	1. Client calls client stub's `grid::_bind()`, which is a static function in the stub. 2. ORB starts a server that contains an object supporting the interface `grid`. 3. As shown in the server main program, Server instantiates all supported objects. (In each constructor, calls are made to create and register an object reference.) Server calls `CORBA::BOA::impl_is_ready()` to tell ORB that it is ready to accept client requests. 4. ORB returns the object reference for `grid` as `gridVar` to the client.
Method invocation	
1. Client calls `pIGrid1->get()` which eventually invokes `CGrid::get()` in the server. 2. To obtain a pointer to another interface `IID_IGrid2` of the same object instance, client calls `pIGrid1->QueryInterface()` which invokes `CGrid::QueryInterface`. 3. When finishing using `pIGrid1`, client calls `pIGrid1->Release()` (which may not invoke `CGrid::Release()` [footnote 1]). 4. Client calls `pIGrid2->reset()` which invokes `CGrid::reset`. 5. Client calls `pIGrid2->Release()` which invokes `CGrid::Release()`.	1. Client calls `gridVar->get()` which eventually invokes `grid_i::get()` in the server. 2. Client calls `gridVar->reset()` which invokes `grid_i::reset()`.

Footnote 1: For performance reason, `Release()` calls for individual interfaces may not be actually forwarded to the server side until all interface pointers that a client holds to the same object are all released. This allows caching interface pointers that may be requested again by the client, and allows lower layers to bundle multiple `Release()` calls in a single remote call.

Figure 3.7: DCOM steps at the top layer

Figure 3.8: CORBA steps at the top layer

is located and activated, and the parties involved in method invocation connect across different processes.

Sending data across different address spaces requires a process called marshaling and unmarshaling. Marshaling packs a method call's parameters (in a client's address space) or return values (in a server's address space) into a standard format for transmission. Unmarshaling is the reverse operation. The packet is unpacked to

Table 3.7: The middle layer description

DCOM	CORBA
Object activation	
7. Upon receiving `CoCreateInstance()` call, COM library delegates the task to Service Control Manager (SCM). 8. SCM checks if a class factory for `CLSID_Grid` has been registered; if not, SCM consults the registry to map `CLSID_Grid` to its server path name, and starts the server. 9. Server registers all supported class factories in a class object table. 10. SCM retrieves from the table the `IClassFactory` pointer to the `CLSID_Grid` factory, and invokes `CreateInstance()` on it. 11. When `CreateInstance()` returns the `IID_IGrid1` pointer, COM (conceptually) creates an object stub for the newly created object instance. 12. The object stub marshals the interface pointer, consults the registry to create an interface stub for `IID_IGrid1`, and associates it with the server object's actual `IID_IGrid1` interface. 13. When SCM ferries the marshaled pointer back to the client side, COM creates an object proxy for the object instance. 14. The object proxy unmarshals the pointer, consults the registry to create an interface proxy for `IID_IGrid1`, and associates it with the RPC channel object connected to the stub. 15. COM library returns to the client an `IID_IGrid1` pointer to the interface proxy as `pIGrid1`.	5. Upon receiving `grid::_bind()` call, client stub delegates the task to ORB [footnote 2]. 6. ORB consults the Implementation Repository to map `grid` to its server path name, and activates the server (in Orbix, the `orbixd` daemon forks the server process). 7. Server instantiates all supported objects, including a `grid` object of class `grid_i`. Class `grid_i` indirectly inherits from `CORBA::Object` whose constructor calls `BOA::create()` with a unique reference ID to get back an object reference. It then registers the object reference with ORB by calling `obj_is_ready()` [Orfali 97]. 8. The constructor for class `grid_i` also creates an instance of the skeleton class. [footnote 3]. 9. When the ORB ferries the object reference back to the client side, it creates an instance of the proxy class and registers it in the proxy object table with its corresponding object reference. 10. Client stub returns to the client an object reference as `gridVar`.
Method Invocation:	
4. Upon receiving `pIGrid1->get()` call, interface proxy marshals necessary parameters, and invokes the `SendReceive()` method on the RPC channel object to send the request. 5. The RPC channel sends the request to the server side, finds the target `IID_IGrid1` interface stub, and calls the `Invoke()` method on it. 6. Interface stub unmarshals the parameters, invokes the method (identified by a method number) on the `grid` object, marshals the return values, and returns from the `Invoke` method. 7. When the RPC channel ferries the marshaled return values back to the client side, the interface proxy returns from the `SendReceive()` call,	3. Upon receiving `gridVar->get()` call, the proxy creates a `Request` pseudo object, marshals the necessary parameters into it, and calls `Request::invoke()`, which calls `CORBA::Request::send()` to put the message in the channel, and waits on `CORBA::Request::get_response()` for reply. 4. When the message arrives at the server, the BOA finds the target skeleton, rebuilds the `Request` object, and forwards it to the skeleton. 5. The skeleton unmarshals the parameters from the `Request` object, invokes the method

Table 3.7: The middle layer description (Continued)

8. unmarshals the return values, and returns them to the client to finish the `pIGrid1->set()` call.	6. (identified by a method name) on the `grid` object, marshals the return values, and returns from the skeleton method. The ORB builds a reply message and places it in the transmit buffer.
9. Upon receiving `pIGrid1->QueryInterface()` call, interface proxy delegates the request to the object proxy's `IUnknown` interface.	7. When the reply arrives at the client side, `CORBA::Request::get_response()` call returns after reading the reply message from the receive buffer. The proxy then unmarshals the return values, checks for exceptions, and returns them to the client to finish the `gridVar->get()` call.
10. The object proxy remotely invokes the actual `QueryInterface()` call on the `grid` object through the same process explained above.	Upon receiving `gridVar->reset()` call, the proxy follows a similar procedure.
11. Upon returning the new `IID_IGrid2` interface pointer, COM creates the interface stub and proxy for it (which share the same object stub and proxy with the `IID_IGrid1` interface stub and proxy, respectively).	
12. The `IID_IGrid1` interface proxy returns to the client an `IID_IGrid2` pointer to the new interface proxy.	
13. Upon receiving `pIGrid1->Release()` call, `IID_IGrid1` interface proxy delegates the request to the object proxy.	
14. Upon receiving `pIGrid2->reset()` call, `IID_IGrid2` interface proxy makes the remote call as usual.	
Upon receiving `pIGrid2->Release()` call, `IID_IGrid2` interface proxy delegates the request to the object proxy which then makes a remote call to release `pIGrid2` (and possibly `pIGrid1`).	

Footnote 2: The stub actually checks its *proxy object table* first to see if it already has an object reference for `grid`. The proxy object table maintains a run-time table of all valid object references on the client side.
Footnote 3: Steps 3 and 4 somewhat correspond to the *implicit activation* policy in POA. POA offers a number of policies related to object activation. Due to lack of space, we will not discuss them in this paper.
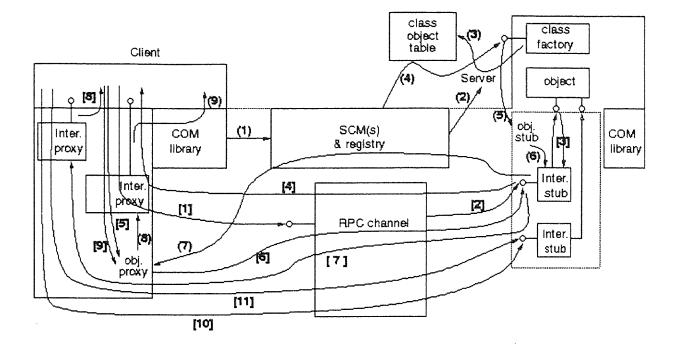
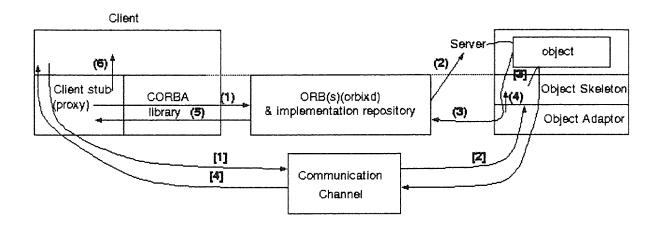**Figure 3.9:** DCOM steps at the middle layer

Figure 3.10: CORBA steps at the middle layer

an appropriate data representation in the address space of the receiving process.

The marshaling process described in the sample application is called standard marshaling

in DCOM terminology. DCOM also provides a custom marshaling mechanism that

bypasses the standard marshaling procedure. (Brockschmidt 93), (COM 95), (Box1 97) Implementing an *IMarshal* interface, a server object declares that it wants to control how and what data is marshaled and unmarshaled, and how the client should communicate with the server. Custom marshaling provides an extensible architecture for plugging in application-specific communication infrastructure (Chung 1997).

The ORB acts as the object bus in CORBA. The Object Adapter (OA) sits on top of the ORB, and is responsible for connecting the object implementation to the ORB. Object Adapters provide services like generation and interpretation of object references, method invocation, object activation and deactivation, and mapping object references to implementations. Different object implementation styles have different requirements which need to be supported by different object adapters. An example of this is object adapters for connection to object oriented databases. The Basic Object Adapter (BOA) defines an object adapter which can be used for most conventional implementations. CORBA specifications not specify how ORB/BOA functionality must be implemented. Iona's built the ORB/BOA functionality into two libraries and a daemon process (orbixd) into its Orbix product. The daemon is responsible for the location and activation of objects. The two libraries, one of which is a server-side library, the other a client side library, are linked at compile time with the respective server and client implementations (Orbix 96).

The (Portable Object Adapter) POA, which was incorporated in CORBA specification 2.2 replaces the BOA. The POA provides portability for CORBA server code and also introduces some new features of the Object Adapter.

The bottom layer of the architectural framework is the wire protocol architecture. The wire protocol supports clients and servers running on different machines connecting through a network interface. The following tables and figures illustrate the steps of connecting clients and server objects over the network..

Table 3.8: The bottom layer description

DCOM	CORBA
Object activation	
16. Upon receiving the delegated `CoCreateInstance()` request, if the client-side SCM consults local registry and finds out that the `grid` object should be located on another server machine, it calls a method of the `IRemoteActivation` RPC interface on the server-side SCM.	
17. When the server is started by the server-side SCM, it is associated with an object exporter and assigned an object exporter identifier (OXID). The mapping from the OXID to the RPC binding that can be used to reach the server is registered with the server-side OXID resolver.	11. Upon receiving the delegated `grid::_bind()` request, client-side ORB consults a locator file to choose a machine that supports `grid`, and sends a request to the server-side ORB via TCP/IP.
18. When the object stub marshals the IID_IGrid1 pointer returned by the CreateInstance(), the pointer is assigned an interface pointer identifier (IPID), unique within the server. Also, an object reference (OBJREF) is created to represent the pointer. An OBJREF contains the IPID, OXID, addresses of OXID resolvers (one per protocol), etc.	12. When the server is started by the server-side ORB, a grid object is instantiated by the server, the CORBA::Object constructor is called and BOA::create() is invoked. Inside the BOA::create(), BOA creates a socket endpoint, the grid object is assigned a object ID, unique within the server, an object reference is created, that contains the interface and the implementation names, the reference ID, and the endpoint address. For clients talking the IIOP protocol, the server generates an **Interoperable Object Reference (IOR)** that contains a machine name, a TCP/IP port number, and an object_key. The BOA registers the object reference with the ORB.
19. When the marshaled interface pointer is returned to the client side through the server-side and client-side SCM's, the object proxy extracts the OXID and addresses of OXID resolvers from OBJREF, and calls the `IOXIDResolver:ResolveOxid()` method of its local OXID resolver.	13. When the object reference is returned to the client side, the proxy extracts the endpoint address and establishes a socket connection to the server.
20. The clients-side OXID resolver checks if it has a cached mapping for the OXID; if not, it invokes the `IOXIDResolver:ResolveOxid()` method of the server-side OXID resolver which returns the registered RPC binding.	
21. The client-side resolver caches the mapping, and returns the RPC binding to the object proxy. This allows the object proxy to connect itself and the interface proxies that it creates to an RPC channel that is connected to the object	

Table 3.8: The bottom layer description (Continued)

exporter.	
Method invocation	
17. Upon receiving `pIGrid1->get()` call, the interface proxy marshals the parameters in the **Network Data Representation (NDR)** format [DCE 95].	8. Upon receiving `gridVar->get()` call, the proxy marshals the parameters in the **Common Data Representation (CDR)** format [CORBA 95].
18. The RPC channel sends the request to the target object exporter identified by the OXID-resolved RPC binding.	9. The request is sent to the target server through the established socket connection.
19. The server-side RPC infrastructure finds the target interface stub based on the IPID that is contained in the RPC header.	10. The target skeleton is identified by either the reference ID or object_key.
20. After invoking the actual method on the server object, the interface stub marshals the return values in the NDR format.	11. After invoking the actual method on the server object, the skeleton marshals the return values in the CDR format.
21. Upon receiving the delegated `pIGrid1->QueryInterface()` call, the object proxy invokes the `IRemUnknown::RemQueryInterface` method on the OXID object [footnote 4] in the target object exporter. The OXID object then invokes the `QueryInterface()` method on (possibly multiple) interfaces within the exporter.	
22. Upon receiving the delegated `pIGrid2->Release()` call, the object proxy invokes the `IRemUnknown::RemRelease()` method on the OXID object in the target object exporter. The OXID object then invokes the `Release()` method on (possibly multiple) interfaces within the exporter.	

Footnote 4: There is one OXID object per object exporter. Each OXID object supports an `IRemUnknown` interface consisting of three methods: `RemQueryInterface()`, `RemAddRef()`, and `RemRelease()`. These methods allow multiple remote `IUnknown` method calls destined for the same object exporter to be bundled to improve performance. All such calls are first handled by the OXID object, and then forwarded to the target interface. Note that these and other bottom-layer APIs are essentially implementation details. Application programmers will not encounter them.
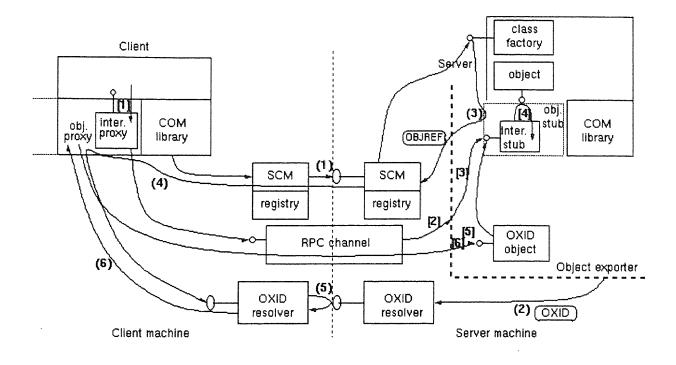
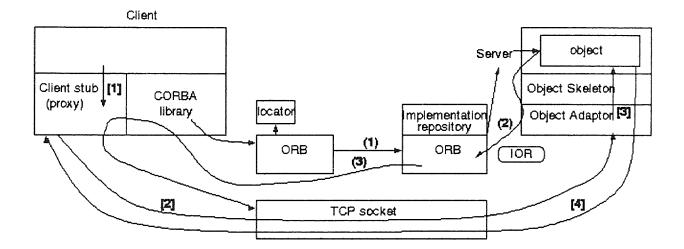**Figure 3.11:** DCOM steps at the bottom layer

Figure 3.12: CORBA steps at the bottom layer

The main differences between DCOM and CORBA at the bottom layer include how

remote interface pointers or object references are represented, and the standard format in

which data is marshaled for transmission in a heterogeneous network environment. It

should be noted that CORBA does not specify a protocol for communication between the client and server running on ORBs made by the same vendor. The protocol for this communication is vendor specific. Interoperability between ORBs from different vendors is specified in the section on the General Inter-ORB Protocol (GIOP). A specific mapping of the GIOP on TCP/IP connections is defined, and is known as the Internet Inter-ORB Protocol (IIOP) (OMG 1998).

The DCOM wire protocol is based primarily on the OSF DCE RPC specification (DCE 95) with a few added extensions. These extensions include remote object reference representation, an *IRemUnknown* interface used for optimizing performance of remote *IUnknown* method calls, and the delta ping protocol (Brown 98). Pinging, discussed in the section on the DCOM architecture, is used by a server object to garbage-collect remote object references when a remote client abnormally terminates. When a client obtains an interface pointer to a remote object for the first time, the ping client code on the client machine adds the object to a ping set and periodically sends a ping to the server machine to let the server object know that it is still alive. If a predetermined number of pings are missed the server object assumes that the client is no longer alive and therefore releases any interface pointers that it holds. To optimize performance, are collected into sets and sent on a per-machine basis and in an incremental way. These pings can also be "piggy-backed" on normal network messages (Brown 98).

3.4 Comparison of the Strengths and Weaknesses of CORBA and DCOM in the Windows Environment

The major differences between the DCOM and CORBA architectures fall into three categories. First DCOM supports multiple interfaces, while CORBA supports a single

interface. Second every CORBA interface inherits from *CORBA::Object* the constructor which performs objects registration, reference generation, skeleton instantiation, etc. All of these tasks are performed explicitly by the Server programs in DCOM or are handled by the operating system. Third DCOM's wire protocol is tied to RPC, while CORBA's is not. Instead CORBA relies upon IIOP or GIOP to handle communication between a client and a server object or between two ORBs.

Another more obvious difference between the two architectures is the fact that CORBA implementations rely upon an addon ORB which installs over the Windows operating system. CORBA provides operating system transparency which is a plus from a portability standpoint. Objects can be developed and moved from one platform to another regardless of the operating system and installed without recompiling provided the target platform has the same vendor supplied ORB installed. The downfall of an operating system independent architecture is that it is much more difficult to interact with other objects on the machine that are not instantiated by the CORBA ORB. An example of this would be a multi-tier client server application which integrates data from more than one source.

DCOM, on the other hand is implemented by the Microsoft Transaction Server (MTS), which is part of Internet Information Server 4.x which is available as an add-on for Windows NT 4.x and Windows 95/98. Once installed, MTS becomes an extension of the operating system since Windows NT 4.x, and Windows 95/98 are built upon the Common Object Model framework. All objects on these platforms can be exposed by the MTS system.

DCOM server objects like CORBA server objects, can be ported from one machine to another. In the case of DCOM, MTS must be installed on the target machine. The difference lies in the fact that each CORBA vendor has a different ORB implementation requiring the developer of server objects to code specific extensions for the specific ORB each having different language mappings and IDL constructs. In the case of DCOM you have one monolithic vendor, Microsoft. Microsoft further simplifies the development process through its Visual Studio 6.0 development suite which includes a C++, Visual Basic, and Java development environment. With these products COM servers or COM wrapped clients can be produced within the integrated development environment. Active Server Pages, HTML pages with scripting languages such as VBScript or JavaScript, can also be exposed as server objects, by MTS. This facility is not found in current CORBA implementations.

CHAPTER 4

CONTRIBUTIONS

A Practical Guide for Choosing Distributed Architectures

The decision on which distributed architecture to use to build an n-tier application is largely one of preference or familiarity. In many instances either architecture could effectively be used to provide a solution. Possible criteria that should be considered include: the server platform, location transparency, and choice of programming language.

DCOM objects, which are closely integrated with the Windows registry, can only be hosted on Windows NT computers which have Microsoft Transaction Server installed. CORBA objects, in contrast, can be hosted on any computer that has a CORBA ORB installed. DCOM components are machine dependent due to their integration with the registry. The same component compiled on two different machines will have two different CLSIDs, which means if components are moved from one machine to another, client applications must be recompiled as well as server objects because DCOM clients transmit the CLSID or UUID (Universal Unique Identification Descriptor) to the machine hosting the server objects. The naming service then uses that CLSID to locate the server object in the registry.

CORBA has many features that foster location transparency and portability. CORBA objects are not dependent on the underlying operating system of the machines they reside on. Instead CORBA implementations are ORB dependent. Because of this CORBA objects can easily be moved from one machine to another assuming that both machines have the same ORB installed. CORBA also has the DII (Dynamic Invocation Interface) which allows client applications to discover and implement server object

methods at run time. One advantage this has is ease of maintenance. Developers can reengineer server object methods without impacting client applications due to the fact that the client's due not have to have the server stub to implement it's methods.

DCOM has the advantage of ease of scalability. Server Object code written for any ORB is to an extent proprietary, because each vendor has different IDL mappings for their ORBs. Due to the that Windows NT has such a large user base developers may make the choice of creating a monolithic codebase for deploying an "enterprise" application using DCOM rather than trying to build a large application on machines that may have ORBS from different vendors requiring server object code written for each target ORB.

Both distributed architectures allow have mapping which allow developers to use different programming languages to produce server objects. CORBA offers mappings for more languages than DCOM to include C, C++, Java, Smalltalk, and ADA. DCOM; however, is the only distributed architecture that has mappings for Visual Basic, or the scripting languages VBScript or JavaScript.

CHAPTER 5

FUTURE WORK AS A CONTINUATION OF THIS THESIS

Possible areas for future research on DCOM and CORBA include DCOM

implementations on non-Windows platforms and interoperability between CORBA and

DCOM. As stated earlier in this thesis, DCOM is currently only available for the

Windows platform. There is development; however, by some vendors such as SAG to

port DCOM to Unix. The ability to develop DCOM server objects on Unix will make

DCOM a viable option for cross-platform distributed application development. Current

implementations of DCOM on Unix can be categorized as remote DCOM objects which

are instantiated on a Unix machine, but controlled by the run time environment of a

Microsoft Transaction Server residing on a Windows machine.

Integration of DCOM and CORBA is might prove to be the optimal solution for

distributed applications in a heterogeneous environment. To facilitate this, a common

strategy for object instantiation, and run time control would have to be developed and

standardized. For example how would a DCOM object which requests data from a

CORBA object know if the if the object is still alive since the methods which maintain

the status of the object's states are different in the two architectures?

CHAPTER 6

CONCLUSION

We have discussed the architectures of both DCOM and CORBA as well as how they are implemented. Both are viable architectures for distributed application development. CORBA is the only current alternative for development on non-Windows platforms. As before mentioned, DCOM is being ported to non-Windows platforms such as Unix. The scope of this document; however, is the Windows environment. Our belief is that DCOM provides a simpler, more powerful, and robust solution for distributing application development on Windows machines. DCOM also provides an easier path to distributed application development for Windows developers already familiar with the COM architecture. We further quantified this statement by narrowing it down to development of server objects since clients applications on any platform can connect to server objects in both DCOM and CORBA provided they contain the appropriate client stub IDL mappings. We compared the relative strengths and weaknesses of the two architectures when implemented in the Windows environment.

We provided general guidelines to consider when choosing an architecture for distributed application development on the Windows platform. Future work should focus on the extension of the DCOM architecture to non-Windows platforms and DCOM/CORBA integration.

REFERENCES

1. A. Birrell, and B.J. Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, vol. 2, no. 1, pp. 39-59, 1984.

2. D. Box, "Q&A ActiveX/COM," *Microsoft Systems Journal*, pp. 93-105, Mar. 1997.

3. D. Box, "Q&A ActiveX/COM," *Microsoft Systems Journal*, pp. 93-108, July 1997.

4. K. Brockschmidt, *Inside OLE*, Microsoft Press, Redmond, WA, 1993.

5. N. Brown, and C. Kindel, *Distributed Component Object Model Protocol – DCOM/1.0* [Online], Available: http://search.ietf.org/internet-drafts/drafts-brown-dcom-v1-spec-03.txt, Jan. 1998.

6. D. Chappell, *Understanding ActiveX and OLE*, Microsoft Press, Redmond, WA, 1996.

7. E. P. Chung, Y. Huang, S.Yajnik, D. Liang, J. C. Shih, C. Wang, Y.Wang, *DCOM and CORBA Side by Side, and Layer by Layer*, [Online], Available: http://www.research.microsoft.com/os/ymwang/papers/HTML/DCOMnCORBA/S.html, Sept. 3, 1997.

8. *Component Object Model Specification*, [Online], Available: http://premium.microsoft.com/isapi/devonly/prodinfo/msdnprod/msdnlib.idc?theURL=/msdn/library/specs/tech1/d1/s1d137.htm, Oct. 24, 1995.

9. *Common Object Request Broker: Architecture and Specification, Revision 2.2*, [Online]. Available: http://www.omg.org/corba/c2indx.htm, July 12, 1998.

10. *DCE 1.1: Remote Procedure Call Specification, The Open Group*, [Online]. Available: http://www.rdg.opengroup.org/public/pubs/catalog/c706.htm, Oct., 1997.

11. R. Grimes, *Professional DCOM Programming*, Wrox Press, Olton, Birmingham, Canada,1997.

12. *Iona Orbix 2.3 Programming and Reference Guide*, [Online]. Available: http://www.iona.com/products/orbix/orbix/fordevelopers.html, 1997.

13. R. Jennings, S. Gray, and R. Lievano, *Database Workshop – Microsoft Transaction Server 2.0*, Sams Publishing, Indianapolis, IN, 1997.

14. K. Keahey, *A Brief Tutorial on CORBA*. [Online]. Available: http://www.cs.indiana.edu/hyplan/kksiazek/tuto.html, 1998.

15. Object Management Group, *What is CORBA????*, [Online]. Available: http://www.omg.org/about/wicorba.htm, 1998.

16. R. Orfali and H. J. Edwards, *Instant CORBA*, John Wiley & Sons, Inc., New York, NY, 1997.

17. D. Rogerson, *Inside COM*, Microsoft Press, Redmond, WA, 1996.

18. D. Schmidt, *Overview of CORBA*. [Online]. Available: http://www.cs.wustl.edu/~schmidt/corba-overview.html, Aug. 1998.

19. D. Schmidt and S. Vinoski, *Object Adapters: Concepts and Terminology (Column 11)*. [Online]. Available: http://www.cs.wustl.edu/~schmidt/C++-report-col11.ps.gz, Dec. 1997.

20. S. Vinoski, *CORBA: Integrating diverse applications within distributed heterogeneous environments, in IEEE Communications*, vol. 14, no. 2. [Online]. Available: http://www.iona.com/hyplan/vinoski/ieee.ps.Z, Feb. 1997.

21. Y.M. Wang, *Introduction to COM/DCOM*. [Online]. Available: http://akpublic.research.att.com/~ymwang/slides/DCOMHTML/ppframe.htm, 1997.

22. Y.M. Wang, *COM/DCOM Resources*. [Online]. Available: http://akpublic.research.att.com/~ymwang/resources/resources.htm, 1997.