An Architectural Comparison of Distributed Object Technologies

by

Jay Ongg

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 19, 1997

Signature of Author .... May 1997 ....
Department of Electrical Engineering and Computer Science
May 23, 1997

Certified by .... May 97 ...
Nishikant Sonwalkar
Director, Hypermedia Teaching Facility
visor

Accepted by ....
ith
Chairman, Department Committee on Graduate Theses

An Architectural Comparison of Distributed Object Technologies
by
Jay Ongg

Submitted to the
Department of Electrical Engineering and Computer Science

May 19, 1997

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

As developers develop larger systems, distributed object technologies came into the foreground to handle complexity. This thesis examines popular architectures of component software which can be scaled to handle large scale distributed systems. These architectures include OLE by Microsoft, CORBA by the Object Management Group, and Java Beans by Sun. Many issues need to be examined, such as what platforms will the system run on and the size of the network the system will run on. The decision of what architecture is suitable for a given complex system is crucial. This thesis provides a comparative analysis to provide a rational basis for selection.

Thesis Supervisor: Nishikant Sonwalkar
Title: Director, Hypermedia Teaching Facility

TABLE OF CONTENTS

1. Introduction and Background

The point of having distributed object technologies is to write powerful yet maintainable applications. The "distributed" makes the system powerful, allowing scaleable solutions that can overcome single machine performance limits as well as geographical boundaries. The "object" makes the system maintainable, allowing extensive software reuse. An organization can use distributed object technologies to gain an advantage in the market while at the same time building a foundation for future applications. This introduction first looks at what an "object" is, and the issues involved in distributing them. The rest of this thesis will describe existing technologies as well as a comparison of them.

1.1 Component Software

In a computer, the hardware performs relatively simple operations, such as OR and XOR. Constructing user-level applications by manipulating these primitives directly is nearly impossible for a human being, so there was a need for high-level languages, such as Pascal, C, LISP, and Algol. Compilers for these languages attempted to abstract the programmer from the low-level microprocessor details. As a result, the complexities of the executing hardware system becomes less of a factor.

However, complexity in software is now becoming a problem. There are trends in modern software requirements:

1. Developers are taking on more complex projects. Examples include the Therac-25 radiation treatment system [LT] and the Patriot anti-missile system. Computers are not just used in code-cracking anymore, but are also used in systems where a failure can be lethal.
2. People are requiring distributed systems. The decentralization of systems is often a tradeoff in maintainability and scalability. An example is the Denver International Airport Automated Baggage system, which ran $55 million over budget and continues to be unreliable [Swartz]. In addition, these systems often contain different platforms and require conformance to legacy systems.

The main problem is complexity. So just like software was used to eliminate the complexities of hardware, a new technology has emerged, still in its infancy, to reduce the complexity in writing software. Perhaps a misnomer, this technology is called *component software.*

The lines between component software and classical software are not as clean cut as those between software and hardware. Component software *is* software, but such software follows a specific architecture that allows distribution and interoperability with other components.

1.2 Objects

One way of designing component software is with object-oriented technology (OOT). An object is a data type, similar to structures in C. The difference is that:

1. Objects can contain functions (known as methods) in addition to variables (known as members).

2. Objects can hide members and methods from a caller.

Depending on who one talks to (*e.g.* Microsoft or the rest of the world), "object-oriented" may mean different things. OOT, according to mainstream thought, is built on top of *encapsulation*, *polymorphism*, and *inheritance*.

1.2.1 Encapsulation

A well-designed object will permit only itself to manipulate its internal data, and in addition will not haphazardly manipulate data structures outside its scope of functionality. This is the main idea of encapsulation.

The "internal data" refers to private member variables and methods, which outside clients can not access. Outside entities can only interact with the object through its public members and variables. Thus, the details of the implementation are abstracted away from the user. The object provides an interface which the caller can use. The interface is a group of methods which act as a contract between the caller and implementer of the object. In the paradigm of object oriented programming languages like C++ and Java, one object provides one interface. In the architecture defined by OLE, an object may provide more than one interface.

1.2.2 Polymorphism

Suppose an air traffic controller wants to tell Planes A, B, and C to land. He doesn't need to know what models the planes are. Each plane knows how to land itself (think of a plane as an object, forget the fact that inside there is a pilot controlling it and other details). The air traffic controller doesn't need to know that A is a Boeing 747, B is a Concorde, and C is an Airbus. He just knows that they are all planes, and all planes have the ability to land, so all he has to do is say, "Plane A, land on airstrip #24", etc.

This is the idea of polymorphism. The caller of a method won't need to know the type of the object the method belongs to. It just knows that the method will perform the appropriate action according to its type. A common way of implementing polymorphism (the only way in some languages) is through the use of inheritance, explained below.

1.2.3 Inheritance

Traditionally, objects in an OO system can *inherit* attributes and methods from another object. This provides a convenient form of software reuse; programmers can create an object by inheriting from a previously-created object. They do not even need the source code to the object they are inheriting from.

Inheritance can be viewed as an *is-a* relation, as in "a circle *is a* shape". However, the idea of instantiation is often viewed as an *is-a* relation. This brings into question just one of the fundamental issues of inheritance. The philosophy of inheritance is a lot deeper than one may think at first glance.

There are many uses of inheritance, but Microsoft has omitted direct inheritance from its OLE architecture. This has been a point of criticism for many purists, but Microsoft had decided to provide the inheritance functionality through other means.

Inheritance is an aspect of OOT that needs to be looked at more closely, because Microsoft's OLE eschews it.

> Kraig Brockschmidt, a member of the OLE design team, wrote: "*Inheritance is a means to polymorphism and reuse*--inheritance is not an end in itself. Many defend inheritance as a core part of OOP, but it is nothing of the sort. Polymorphism and reuse are the things you're really after." [Brockschmidt]

Microsoft implemented reuse and polymorphism in OLE through other means, and Brockschmidt's response implies that those two features are the only reason why inheritance is necessary. This thesis examines these object technologies' methods of reuse in Section 5.3, Reusability.

1.3 Out-of-process Objects

When the first object-oriented programming languages were designed and implemented, designers only thought that clients would invoke methods on objects that were local to the client. Therefore, object oriented technology evolved with this view of in-process objects; programmers never thought about where the objects existed, and millions of lines of code were written without taking this into account. This is, along with simplicity, is one reason why transparency of distribution is important.

In the 1980's, the Open Software Foundation (OSF) started its Distributed Computing Environment (DCE) research, and designed the Remote Procedure Call (RPC). It created a platform independent and transport independent method of calling procedures running in different processes. The remote procedures do not necessarily have to be running on machines with the same architecture as the calling process; thus it enabled the proliferation of heterogeneous networks. DCE RPC was not designed to work with objects, though. Microsoft's DCOM (described in Section 3), however, uses a superset of the DCE RPC wire protocol for its method invocations; they call it Object RPC (ORPC). The Object Management Group, in deciding what to use for IDL in describing objects, considered using DCE RPC's IDL as a basis (like Microsoft).

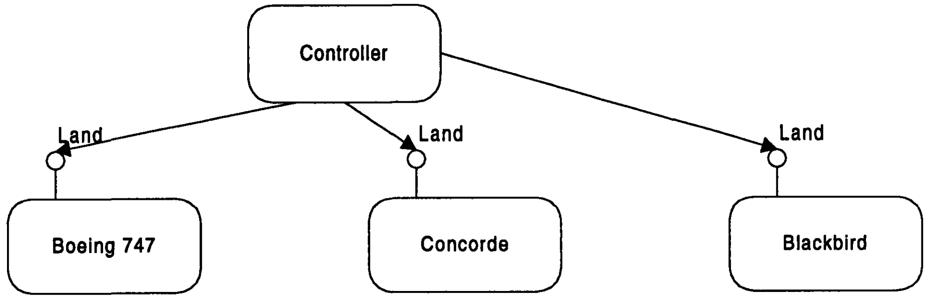


Figure 1 Polymorphism. The Controller does not need to know anything about the model of each plane; he just needs to know that they can all Land.

However, they decided to design their own unencumbered IDL instead. Many CORBA vendors, however, build on top of RPC for object communication.

1.3.1 Local Out-of-process Objects

When an object is on the same machine as the client, but in a different process, there are many issues that every method invocation needs to address. When a method is invoked wbith parameters, the data must be marshaled and transmitted to the process the object exists in. The server process must then unmarshal the arguments and invoke the method. Similar marshaling/unmarshaling must be done with any return values.

In C++ and C, many data structures contain pointers to parts of memory (buffers or other data structures). Such data structures have to be specially marshaled, the designer of the object needs to write or generate marshaling code. The burden of this falls on the object implementer, not the user.

On the same machine, two processes can share memory; this is faster than marshaling and unmarshaling data. Some CORBA vendors use this method to share data locally.

Objects also need to pass around other objects, not just unencapsulated data. Objects are usually not copied around if they are mutable or not explicitly copied. The reason is that changes made to one object may need to be reflected in one single location so that other clients may be able to access the changed object. Since objects can only be passed by reference, each object needs a particular object reference to distinguish it from other instances.

These are some of the issues that need to be addressed in out-of-process objects.

1.3.2 Remote Objects

The issues that affect local servers also affect remote objects. Since remote objects reside on other machines, method invocations need to work over a network. Network lines may fail and then work again; it is up to the distributed object infrastructure to detect when remote servers become unavailable and reconnect to them if necessary. Different strategies work best in different situations.

Once there are remote objects, the number of objects possible increases by orders of magnitude. Therefore each object infrastructure should have a way to detect how objects are located and their interfaces obtained. Since distributed object technologies are only recently being deployed in large quantities, many shrink-wrapped object technology systems are very limited in managing these systems. As of now, no distributed object technology infrastructure provides adequate support for managing large graphs of objects; companies need to develop or contract such management software [Newman]. Many CORBA venders provide adequate dynamic naming services that allow clients to choose which objects to use. Richer trading services should come out at the end of 1997 [OHE2].

Finally, clients will have to be written to make use of these trading services and dynamically bind to them. When Rapid Application Development tools like Visual Basic and Delphi came out, they provided a strong glue for component software. If these components can reside on other servers, then the reach and power of these programs increases tremendously.

1.4 Distributed Object Technologies

With a defined description of objects, it is necessary to see how they can be used by programmers. This thesis does not look at or compare object-oriented languages like Smalltalk, C++, or Java. Distributing objects requires more than just a language; it needs a whole architecture.

Effectively allowing remote objects is not as simple as it sounds. Providing the capability to distribute objects is only part of the problem; effective distributed object architectures need to provide a good infrastructure for managing those objects effectively. Architectures need to provide a good set of services and capabilities that make object distribution as simple as possible. The rest of this thesis will examine and discuss the merits of various technologies. These technologies include the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA), Microsoft's Distributed Common Object Model (DCOM), and Java solutions.

2. CORBA Specification

The Object Management Group (OMG) was formed in 1989 to handle the growing complexities of software development. It creates and publishes specifications that allow interoperability and hasten the development of standardized distributed object software. Currently it has more than 700 members.

The OMG was the group that created the Common Object Request Broker Architecture (CORBA). CORBA is only a specification, allowing venders to implement the architecture in any way they wish.

2.1 Overview

The main goal of CORBA is to allow interoperability between objects on distributed systems. With the newest specification of CORBA, version 2.0, different vendors can now communicate with each other, creating the "intergalactic object bus". Thus, CORBA provides a good architecture for working with distributed objects on a heterogeneous network; this aids greatly in integration of legacy applications.

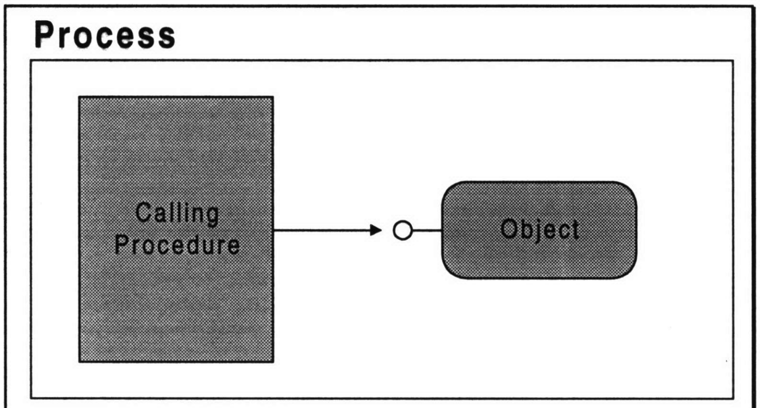


Figure 2 A traditional method invocation where the object is in the same process as the caller

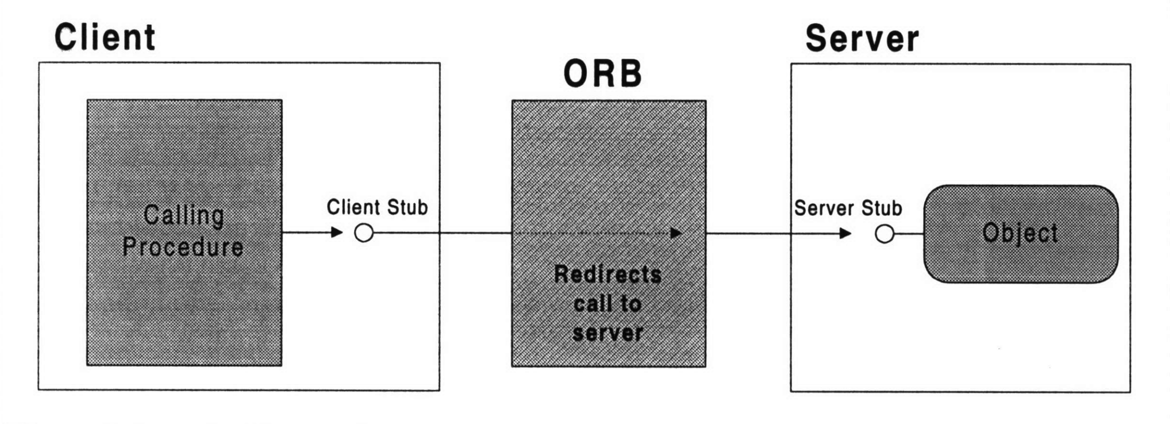


Figure 3 A method invocation using CORBA (different processes)

CORBA works by allowing clients and servers to communicate without worrying about the network protocol and other communication aspects. In this description of CORBA, the "client" is the process that makes use of an object and the "server" is the process that contains the instantiated object. The client does not need to know where the server is; the client can work just as if the object it is working with exists in the same process space. A CORBA implementation achieves this by creating client stubs and server skeletons; they marshal and unmarshal requests and responses. Transparency was an important design goal of CORBA.

Figure 2 and Figure 3 illustrate the difference between a traditional method invocation and one that uses an ORB. Note that the client and server do not know the whereabouts of the other; the ORB handles it all.

The next section talks about various parts of the architecture in more detail.

2.2 Architecture

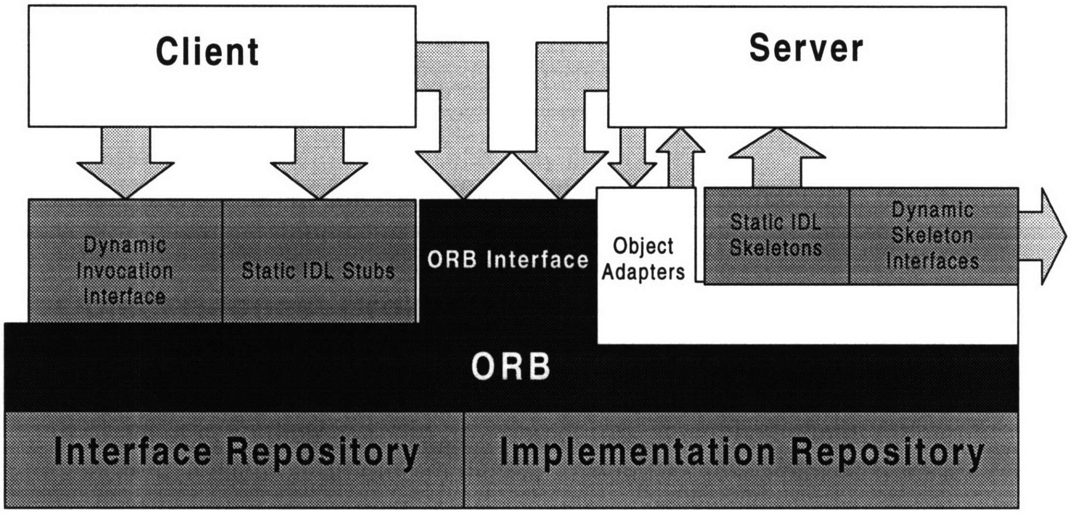


Figure 4 Architecture of CORBA

The client and server are connected by the Object Request Broker (ORB). The ORB does much of the work, taking the communications work from the client and server. The object must first be described not in the language it is implemented in, but in an Interface Description Language (IDL). The IDL specifies a contract between the caller and implementer of the object. This way the implementer and user of the object do not have to be the same person. An interface in CORBA represents a group of methods and members variables. The member variables may also be read-only.

Figure 4 is a more detailed diagram of the workings of the ORB than Figure 3. There are more visible components to the ORB in this diagram.

2.2.1 Object Request Broker (ORB)

The Object Request Broker (ORB) provides the "glue" between the client and server. It directs requests and replies between clients and servers. The specification does not dictate how to implement it, so there is room for competitors; it can be an integral part of the operating system, a daemon running in the background, or just a pair of libraries.

To help it with its task, the ORB has two main parts - the Interface Repository and Implementation Repository.

2.2.2 IDL

The Interface Description Language (IDL) is a purely declarative language that specifies the interface that an object should implement and a client must use. It is a superset of a subset of C++. But, since it is declarative, a user cannot write a functioning program in IDL.

The IDL declarations map into a programming language in a standard OMG-decided way, so a client and server can be "ported" to other CORBA implementations with little or no code change. Official mappings currently exist for C, C++, Smalltalk, and Ada95. In progress are mappings to Java and COBOL. In the rush to jump on the Java bandwagon, some companies have already implemented their own mappings to Java.

```
interface Car {
        attribute float speed;
        attribute float direction;
        void drive(in float distance);
};

interface Police_Car : Car {
        attribute boolean siren_on;
};
```

Figure 5 *myprog.idl*, an example of an IDL file.

The IDL file is used to generate stub files for the client and server; it marshals parameters and sends them to the ORB. Thus, the client and server do not need to know location information; allowing the programmer to concentrate on the object implementation, not network handling.

`myprog_c.hh`	Header file for use with the client
`myprog_c.cc`	Implementation of client stub code
`myprog_s.hh`	Header file for use with the server
`myprog_s.cc`	Implementation of server skeleton code

Table 1 Files generated by Visgenic's IDL compiler

Figure 5 is an example of an IDL file. *Car* is an interface. It contains two attributes and a method. The attributes, *speed* and *direction*, map to get/set methods. The *Police_Car* interface inherits from *Car*, meaning that it can use the same functions as *Car.* Also note that since attributes map to methods, setting *siren_on* = TRUE may invoke a method. This is a convenient way for a user to activate the siren, for instance.

When an IDL compiler for C++ processes it, it produces files similar to those in Table 1. The client stub and server skeleton files are to be linked into the client and server programs, respectively.

Figure 6 depicts the role of the IDL compiler. When a client invokes a method, a method from **`myprog_c`** is actually invoked. This stub method marshals the

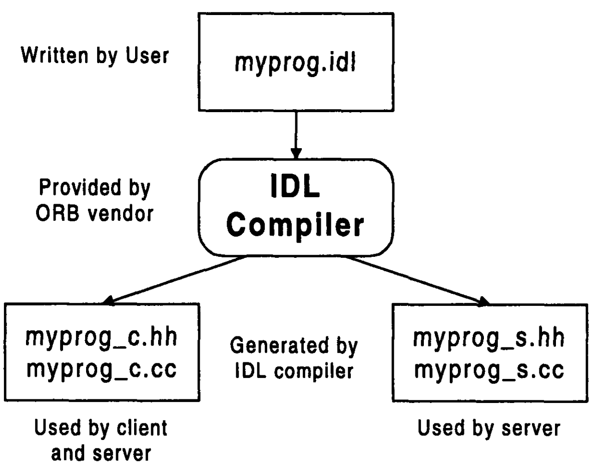


Figure 6 Role of the IDL compiler

parameters and tells the ORB to pass them along to the server. The server skeleton **`myprog_s`** then receives the call, and passes it along to the real object. The response is handled in a similar fashion.

2.2.3 Object Adapters

There are so many ways to implement and object that there must be a standardized way for the ORB to work with the object. This is provided by the Object Adapter. They are responsible for:

- Registering implementations
- Generating and interpreting object references
- Mapping object references to their corresponding implementaions
- Activating and deactivating object implementations
- Invoking methods
- Coordinating interaction security

[Siegel 78]

The OMG has standardized one Object Adapter, called the Basic Object Adapter (BOA). There are other Object Adapters in the standardization process right now. Going into the details of the Basic Object Adapter is beyond the scope of this thesis.

2.2.4 Interface Repository (IR)

The Interface Repository (IR) is an important part of the ORB that contains type information for interfaces. The IR requires a form of persistent storage to store the interfaces. The information in an IDL file is sufficient to be stored in the IR. However, an object can add, modify, or delete an interface in the IR during runtime. This opens up the field to allow dynamically constructed invocations, described below.

The IR is necessary in the use of the Dynamic Invocation Interface, which allows invocations to be constructed during runtime. This will be explained below.

Note that even though the Implementation Repository and Interface Repository both have the same initial letters, IR is used to denote the latter.

2.2.5 Implementation Repository

The Implementation Repository is a part of the ORB that contains information on where to locate objects as well as which objects are currently instantiated. A system administrator can register objects manually, or a server can register objects during runtime. The implementation repository is used by the ORB when a client wants to connect to a particular object or activate a new one.

2.2.6 Dynamic Invocation Interface (DII)

As mentioned above, the Dynamic Invocation Interface (DII) is used to create invocations during runtime, as opposed to using client stubs (note that using client stubs to invoke a method uses the Static Invocation Interface, or SII). This allows a client to use an interface that it was not compiled with. A user can thus browse through the IR, see any new objects he might want to use, and then invoke a method.

Using DII invocations is more complex than using the SII. So in general, the DII should be used only when necessary; when interface information is not available. One common place to use the DII is in scripting languages. This all happens on the client side; a server can not tell whether an invocation was made using the SII or DII.

To aid in scripting, dynamically invoked methods can have named value pairs. This integrates well with many popular macro languages, including Microsoft Visual Basic. Thus, it is possible to call a function with something like:

```
Address = Lookup(ID = 12345, Name = "Jay Ongg")
```

Similar to OLE Automation (in Section 3.2), the complexities involved in making a dynamic request make macro languages the best medium to use DII.

2.2.7 Dynamic Skeleton Interface (DSI)

The Dynamic Skeleton Interface (DSI) is a new addition to CORBA 2.0. It lets the ORB pass requests to other ORBs or non-CORBA systems. This makes the DSI very important in a heterogeneous network, especially one with legacy systems. One can view the DSI as a "filter" for method invocations.

Similar to the DII, a client can not tell whether the object they are using is accessed using the DSI or not.

2.2.8 Internet Interoperability Protocol (IIOP) and the Intergalactic Object Bus

CORBA 2.0 also required ORB implementers to support the Internet Interoperability Protocol. This protocol works on top of TCP/IP and allows CORBA 2.0 compliant ORBs to communicate with each other. This way, the Internet can be used as the network in which object data is transmitted. This is what CORBA supporters call the "Intergalactic Object Bus". There then becomes no limits on where an object may reside; a client may invoke a method on an object halfway across the globe as easily as it can on an object on the same computer. The system administrators just have make sure the ORBs are configured correctly, and the necessary security precautions are in place.

When a client wants to communicate with an object on another ORB, it simply communicates with its local ORB and the local ORB will forward the request appropriately.

Interfaces to be used outside a local domain will need an identifier that is unique in space and time. Therefore, a human generated name like "MyInterface" is not good enough.

CORBA allows two ways of identifying interfaces: via a human generated name, or via a *Universally Unique Identifier* (UUID). The UUID is a 16 byte number generated by an algorithm that the Open Software Foundation discovered to help its development of the Distributed Computing Environment. The algorithm looks at the current time as well as the MAC number of the computer's ethernet card (if it contains one) to guarantee uniqueness. If the computer does not have an ethernet card, then the algorithm will not guarantee uniqueness, but the chances of collision then become extremely unlikely (i.e. the chances of quantum mechanics affecting the macroscopic world are higher than this algorithm producing a conflict. So it's safe to say that pigs will fly before the algorithm produces a duplicate ID).

2.3 CORBAservices and the Object Management Architecture (OMA)

The infrastructure defined above is sufficient for objects to communicate with each other. However, the OMG wanted to create a fundamental set of interfaces that vendors could implement for users. They defined the Object Management Architecture (OMA), and divided it into two parts: The CORBAservices and CORBAfacilities.

The CORBAservices are still being specified, so not many vendors have implemented them yet. They comprise low level interfaces that should be implemented, and the CORBAfacilities will build on top of them.

Here is a partial list of the CORBAservices:

- **Lifecycle Service:** Provides services and conventions for creating, deleting, copying, and moving objects
- **Persistent Object Service:** Provides services that simplify the creation of persistent objects (i.e. objects that exist when the ORB is shutdown and restarted)
- **Externalization Service:** Provides services, interfaces, and protocols that allow an object to be written as a stream of bytes.
- **Naming Service:** Provides an interface for an object to be bound to a name.
- **Trader Service:** Provides an interface for an object to be bound to a particular category. One may think of the Naming Service as the "White Pages" and the Trader Service as the "Yellow Pages".
- **Event Service:** Provides an interface for an object to send asynchronous events to its client.
- **Transaction and Concurrency Control Services:** Provides services and interfaces that allow transactions and concurrency control.
- **Property Service:** Allows the user to add a property to an already defined and instantiated object. A property is a string binding to any data structure.
- **Security Service:** Provides services that enable security in the distributed object model.

2.4 CORBAfacilities and CORBAdomains

While the CORBAservices create a rich infrastructure that a programmer can use to develop CORBA applications, an infrastructure is not enough for a developer to build component software. With what we've talked about already, a user can not just pick a spreadsheet, attach it to his word processor, and start typing.

The OMG is in the process of specifying the CORBAfacilities, which are these application-level objects. CORBAfacilities are to be implemented using the CORBAservices. The CORBAfacilities are a set of horizontal APIs that have a general purpose use. The CORBAfacilities are divided into four basic areas:

- **User Interface:** For interaction with the user.
- **Information Management:** For managing and interchanging information.

- **Systems Management:** For managing complex, multivendor information systems
- **Task Management:** For automating work processes.

The CORBAdomains will cover specific vertical markets. Currently, the OMG has groups working on interfaces for Healthcare, Telecommunications, Transportation, Financial Services, Manufacturing, Electronic Commerce, and Business Objects. These names are self-explanatory, except for the Business Objects group. This group is working on a standard that will model people, entities, and anything active in the business domain.

3. OLE

OLE is a general term for Microsoft's object-based infrastructure for component software. OLE originally stood for Object Linking and Embedding, the capability for one program to link or embed data for another program. The overused example is that of embedding an Excel table in a Word document; then when the user double-clicks on the Excel table, Excel pops up in a new window.

Then came OLE 2, which enabled *in-place activation*. Now when a user double-clicks on the Excel table, they can work on the Excel table in-place, without opening a new window. OLE 2 was also built on an object model called the Component Object Model (COM). COM is what forms much of the glue between components.

As OLE evolved, Microsoft just got rid of the version number and called the whole architecture OLE. New features were added: OLE Automation, OLE Controls, Local/Remote Transparency, and other services. Anyone can see that Microsoft has created a rich architecture for component software.

Where CORBA was designed with distributability in mind, OLE was not. OLE was an aid in the trend towards the document-centric desktop; the computer shouldn't make the user care about applications, just documents. Microsoft has recently incorporated distribution into OLE, with the advent of the Distributed Component Object Model (DCOM).

Finally, with the explosion in growth of the Internet, Microsoft jumped on the bandwagon with its ActiveX technology. One may have heard about the debate between ActiveX and Java; such a comparison is beyond the scope of this thesis.

One interesting thing about Microsoft and its OLE technology is that they write the specifications (if any) and until recently, were the only implementers. So, in the past, behavior and requirements have changed, to the chagrin of developers everywhere. In fact, with ActiveX, since Microsoft jumped on quickly, there are many undocumented parts of it and there was not even a specification on what defines an ActiveX control.

It is impossible to cover all the aspects of OLE in a Master's Thesis; entire books have been written on fractions of it. The following sections should provide the reader a good background to understand the comparisons of it that will come later.

3.1 Component Object Model (COM) and Distributed COM (DCOM)

The Component Object Model (COM) is the fundamental basis of all of OLE. In fact, the other major parts of OLE (Automation, DCOM, and ActiveX) all build on COM.

CORBA does not care how implementations handle the objects. The requirement is that a CORBA ORB must be accessible through code created with the IDL. COM, however, is a binary specification; it matters very much what the compiled code looks like; COM works with pointers and arrays of pointers. One bad side effect of this is that executables created with Microsoft's Visual C++ will work with COM, whereas those created with other compilers may not.

3.1.1 Interfaces and Classes

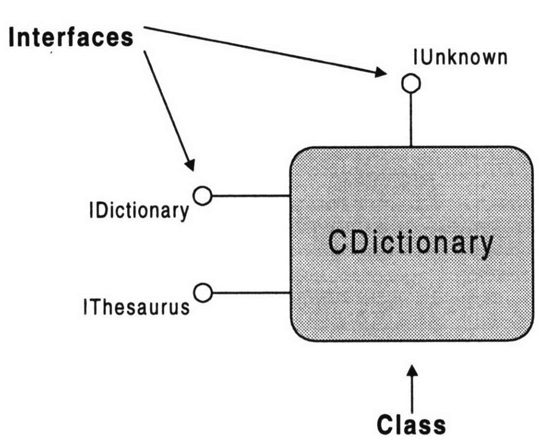


Figure 7 Example of a COM Object

A COM object can be access through an interface. Unlike in CORBA, a COM interface can not hold member variables, just methods. A COM interface is identified by an Interface Identifier (IID). An IID is represented by a UUID (described in Section 2.2.8), which Microsoft renamed as a *Globally* Unique Identifier (GUID).

A COM object is an implementation of one or more interfaces. It is also identified by a GUID, known as a Class Identifier (CLSID).

A COM object can only be manipulated through an interface; a client can not work directly with the class itself. This is one of the fundamental concepts of COM.

That was a quick run-through of various terms used in COM. For a concrete example, look at Figure 7. The class *CDictionary* implements the interfaces *IUnknown, IDictionary*, and *IThesaurus*. Those are all human-readable names; to a computer, *CDictionary*, *IUnknown, IDictionary*, and *IThesaurus* would be represented by 16 byte GUIDs. This is an example of an object that implements more than one interface.

3.1.2 IUnknown and its members

One interface that all COM objects must implement is *IUnknown.* It contains three methods necessary for COM to work: *QueryInterface*, *AddRef*, and *Release.*

QueryInterface allows a client to ask an object whether it implements a certain interface. So, through *QueryInterface*, a client may ask an instance of *CDictionary*, "Do you implement *IDictionary*?" and get a response. *AddRef* and *Release* are used for reference counting; the object is only destroyed when the reference count is 0.

Not only must all COM objects implement *IUnknown*, but every COM interface must also implement *QueryInterface*, *AddRef*, and *Release.* This way, a client can access other interfaces of an object no matter what interfaces it currently knows about.

3.1.3 MIDL and RPC

The architecture that Microsoft uses to marshal arguments and pass them across boundaries is based on the Microsoft's Remote Procedure Call, based on the RPC from the Open Software Foundation's Distributed Computing Environment. Similar to CORBA, COM/OLE and OSF RPC use an IDL language. Microsoft Interface Description Language is often called IDL or MIDL; this document will refer to it as MIDL.

Unlike in CORBA, MIDL is not necessary to create an OLE object; it exists for writing custom parameter marshaling code and for the creation of type libraries. Only a subset of MIDL is used in OLE, since it was designed for broader applications of remote procedure calls. Figure 8 is an example of a MIDL file. This file looks very dissimilar to the IDL file depicted in Figure 5, even though they are describing the same functionality. This thesis will go through this in more detail than with the CORBA IDL because MIDL is not as intuitive. Even though it seems that some of these constructs seem unnecessary, keep in mind that MIDL is used by RPC, not just OLE.

```
import "unknwn.idl";

[uuid(00000000-0000-0000-0000-000000000000),
        object]
interface Car : IUnknown {

        HRESULT GetSpeed([out] float *speed);
        HRESULT SetSpeed([in] float speed);
        HRESULT GetDirection([out] float *dir);
        HRESULT SetDirection([in] float dir);
        HRESULT drive([in] float distance);
};

[uuid(00000000-0000-0000-0000-000000000001),
        object]
interface Police_Car : Car {

        HRESULT GetSiren([out] BOOLEAN *on);
        HRESULT SetSiren([in] BOOLEAN on);
};
```

***Figure 8** Mycars.idl,* An example of a MIDL file.

There are two interfaces here, *Car* and *Police_Car*. Each interface is preceded by a tag that indicates the UUID of the interface, and the fact that the interface is to be used by an object.

In the **`interface`** lines, each interface inherits from *IUnknown* (the *IUnknown* interface is visible because of the import statement at the top of the file). This gives the interface the *QueryInterface*, *AddRef*, and *Release* methods necessary to all COM objects.

Finally, for each of the actual interfaces themselves, not how they are declared. For OLE to use an interface, it must return an HRESULT (which is a 32 bit error return code). Any return values must come through the parameters. Like CORBA, there are in and out parameters.

One thing to note is that even though *Police_Car* inherits from *Car*, this only exemplifies interface inheritance. In general, OLE does not use inheritance to reuse implementations. In OLE, reuse is achieved through multiple interfaces, via aggregation or containment. This will be discussed in more detail in Section 5.3, Reusability.

When the MIDL compiler processes the MIDL file, it generates a few files, described in Table 2. The role of these files is not like that of the files generated by CORBA's IDL compiler. The *MyCars_i.c* and *MyCars_p.c* files are to be linked into both the client and server. To simplify this, the MIDL compiler creates the *dlldata.c* file which contains routines necessary for a functioning DLL. Figure 9 shows the role of the MIDL compiler.

`MyCars.h`	Header file containing *struct* declarations
`MyCars_i.c`	Contains the IID definitions
`MyCars_p.c`	Contains proxy/stub code
`dlldata.c`	Necessary information to create a DLL

Table 2 Files generated by the MIDL compiler

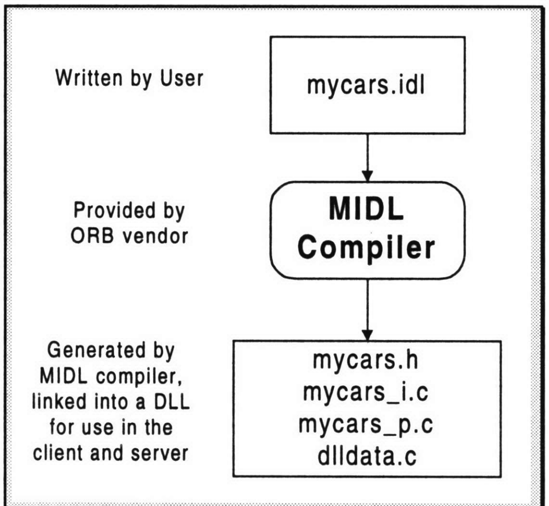


Figure 9 Role of the MIDL compiler

3.1.4 Object Registration

In COM, a client must call a global API function, *CoCreateInstance*, to get an instance of a class. On the Windows platforms, *CoCreateInstance* searches the registry to find a certain CLSID; the registry contains activation information (such as how to launch the server, and where it is). One can view the registry as the COM equivalents of CORBA's IR and Implementation Repository.

Having a system administrator register every COM object into the Windows registry reduces scalability and maintainability. Therefore, there is a shift to creating servers that register their objects on startup. The disadvantage is that the operating system can not launch a server; it must already be running to make its objects available.

3.1.5 Local/Remote Transparency

Like CORBA, COM has a method of accessing objects outside the process of the client application. Like in CORBA, there are rules required in accessing an object; if these rules are followed, then COM can marshal arguments and transmit them for in-process servers, local servers (same machine, different process), and remote servers.

When an interface pointer to a COM object is obtained, it can point to two things:

1. A pointer to the real object itself (if the object is in-process)
2. A pointer to a proxy object that will marshal the arguments over to the server.

Most objects use *standard* marshaling. Standard marshaling is COM's default marshaling routine. However, method invocations on certain classes of objects can be optimized more; COM allows a programmer to do this with *custom* marshaling. Thus, if a developer feels that Microsoft's marshaling method is not optimal for his object (and for immutable objects it is often a waste of network bandwith or interprocess communications), he can rewrite the marshaling code (by implementing a particular interface called ***IMarshal***). Since this is an option, an object developer can implement an object without worrying about the remote procedure calls to it, unless he wants to.

3.1.6 COM Services and Features

Like CORBA, COM provides more than just an infrastructure or rules for working. Although COM and CORBA share many different services, COM was developed on Microsoft Windows, so there is a definite desktop-oriented flavor to the services it provides. Many of these features enable interaction between different applications and contribute to a document-centric operating system. Among them are:

- **Persistence:** Similar to CORBA, COM provides a mechanism for persistent objects.
- **Structured Storage:** COM provides an interface for reading and writing to files. The implementation of this is called *Compound Files*. This feature allows a programmer to write data in an organized fashion to a file; it simulates a file system within a file to facilitate storage and retrieval of data streams.
- **Connection Points:** Arising from Windows' use of callback functions, connection points allow an object to call interfaces on a client easily.
- **Monikers:** After Microsoft's experience in developing desktop applications, they needed a method to keep track of names of linked and embedded objects. They therefore provided and extra level of indirection to facilitate this, and called them monikers.
- **Uniform Data Transfer:** This is an interface that grow from desktop application interactions. Originally, there were different methods that a program had to use to communicate with other programs, the clipboard, embedded objects, etc. Microsoft designed the Uniform Data Transfer interface which objects can implement and clients can call. Now to communicate with different objects, a client can just use this interface regardless of what object implements it.

3.1.7 Distributed Component Object Model (DCOM)

When Microsoft released Windows NT 4.0, they also released an extension to COM called DCOM, for Distributed Component Object Model. DCOM allows applications to work with objects on other computers. This is what brings COM from the desktop to the network. Almost nothing has to be done by a programmer to make a client or object distributed.

However, since DCOM is mainly an extension of COM, little has been done on various distributed issues; such as object announcement, trader services, etc. New features like this will come out in Windows NT 5.0.

3.2 OLE Automation

COM is a good infrastructure for compiled programs that wish to make use of objects. Each interface in COM is compiled into the code, similar to CORBA stubs. COM interfaces do not provide as much flexibility and speed for dynamic invocations, and in addition, not all COM interfaces have type information associated with. Thus, to dynamically invoke a COM interface, there is a lot of complexity involved in obtaining an object's type information.

Because of this, COM is not well-suited for macro language use. In general, macro languages dynamically bind objects during run-time (and in the case of Visual Basic, during editing to aid with programming). Scripting languages need a different way of working with objects, a late-binding mechanism. OLE Automation, which arose from the Visual Basic team, is Microsoft's solution. There is a change in terminology: a client is called a *controller*, since Automation was developed to allow Visual Basic programs to control other applications (such as Microsoft Word).

3.2.1 Dispinterfaces and IDispatch

OLE Automation is built on top of COM; it provides an extra level of indirection. Automation servers export their objects through an interface called a *dispinterface*. Dispinterfaces are similar to standard COM interfaces except that:

1. Dispinterfaces can contain *properties*, which are simply member variables (it is also possible to specify whether they are read/write or read-only.
2. Each method or property in a dispinterface must have associated with it a *dispID*. No two methods or properties in a dispinterface can share the same dispID.

Method	Description
Invoke	Given a dispID and other necessary parameters, calls a method or access a property in this dispinterface.
GetIDsOfNames	Converts text names of peroperties and methods to their corresponding dispIDs.
GetTypeInfoCount	Determines whether there is type information available for this dispinterface, returning 0 (unavailable) or 1 (available).
GetTypeInfo	Retrieves the type information for this dispinterface if GetTypeInfoCount returned successfully.

Table 3 The methods of ***IDispatch*** [Brockshmidt2 643]

Once an Automation server is developed to export a dispinterface, then it must implement the ***IDispatch*** COM interface to use it. This interface is the key interface of OLE Automation. The methods of the ***IDispatch*** interface is shown in Table 3. The controller must use this to access dispinterfaces. How the controller actually makes use of this interface is beyond the scope of this thesis. Note that ***IDispatch*** is *not* a dispinterface, it is a COM interface that enables the server and client to work with dispinterfaces.

When working with OLE Automation objects, it is possible to pass data structures and Visual Basic data types (currency, date/time, etc.). It is not possible to pointers, since they make almost no sense in a macro language. However, starting with Visual Basic 5.0, Microsoft will specify a protocol so that the controller can pass in callback functions to the server.

To simplify the use and manipulation of dispinterfaces, MIDL can also describe them. This lets programmers describe the methods of a dispinterface for use in a type library. Once a type library is made, 4GL tools may make use of them to simplify development.

3.2.2 Locales

One advantage of dynamic binding that OLE Automation takes advantage of is that Automation servers can give their dispinterfaces and methods different names. For localization of Automation controls, this feature is very useful. For a regular COM object, it is not as flexible since so much is programmed into the object itself.

A server can also support multiple locales. "Locale" covers more than just language; there are the issues of date/time formats and currency, too. Programmers tend to want to program in English because all system APIs are in English and all the lower-level programming tools and languages express their capabilities in English. A higher level user, however, would prefer scripting in his native language. An OLE Automation server can allow both groups of users to work the way they want, with the same object. Thus an end-user can drive the object with his personal script, which, at the same time, is driven by a corporate developer's script in English.

In fact, an OLE Automation server can support *every* locale available, but this ideal situation is probably not going to happen.

3.2.3 Remote Automation

Before DCOM came out, Visual Basic 4.0 Enterprise Edition allowed distributability of OLE servers and clients through *Remote Automation.* Remote Automation allows Automation controllers and servers to be physically apart, and in addition lets Automation controllers make asynchronous method invocations. It also aids greatly in making three tiered client-server architectures.

3.3 ActiveX and the Internet

From the technologies of embedded documents and in-place activation of thsese documents arose a type of object called an OLE control. Many OLE controls were also OLE Automation servers; this allowed controls to be scripted.

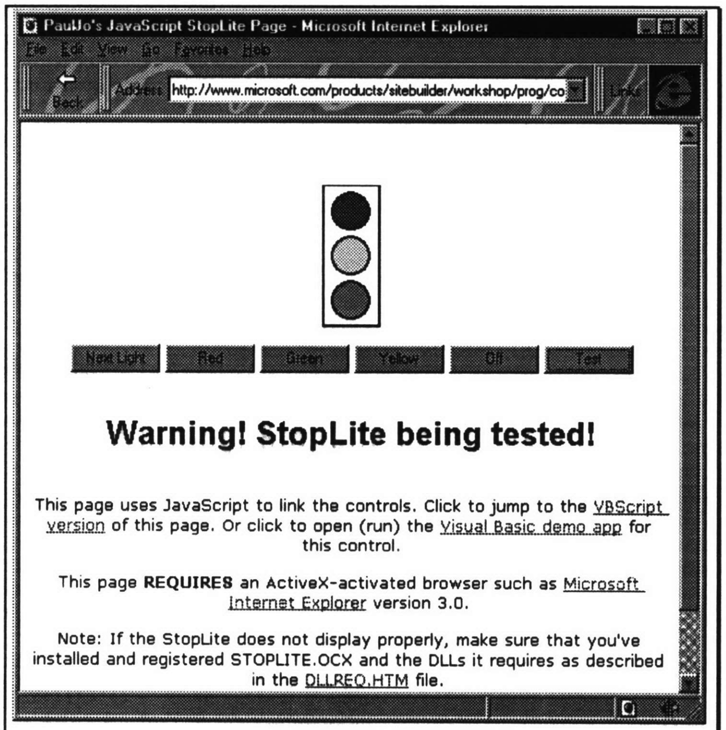
OLE controls were simply objects that could be embedded in a container (such as Visual Basic 4.0 or Microsoft Internet Explorer 3.0). In a standard OLE document embedding (like the Excel into Word cliché), the inner document is activated by the contained. This is called *Outside-In* activation. An OLE control, however, can have *Inside-Out* activation; thus it can send events out to its container.

With the explosion of the Internet, Microsoft decided to leverage its OLE controls for use over the Internet. Most home users have modem speed connections, so Microsoft cut down the number of interfaces required to implement a control, and renamed them ActiveX controls. This section talks about how ActiveX controls are used over the Internet.

3.3.1 Architecture and Example

An ActiveX control is a dynamically linked library that a user can install on his system. These controls implement specified interfaces that a control container can use. The control can fire events that the container can detect; this makes a control most useful in a scripting environment.

Figure 10 is an example of an ActiveX control. The stoplight is an ActiveX control; it interacts with the container (the web browser) through JavaScript, a scripting language. The buttons are not ActiveX controls; they are part of the HTML page which the user uses to interact with the control itself.


PaulJo's JavaScript StopLite Page - Microsoft Internet Explorer
File Edit View Go Favorites Help
Back
Address
http://www.microsoft.com/products/sitebuilder/workshop/prog/co
Links
Next Light
Red
Green
Yellow
Off
Test

Warning! StopLite being tested!

This page uses JavaScript to link the controls. Click to jump to the VBScript version of this page. Or click to open (run) the Visual Basic demo app for this control.

This page **REQUIRES** an ActiveX-activated browser such as Microsoft Internet Explorer version 3.0.

Note: If the StopLite does not display properly, make sure that you've installed and registered STOPLITE.OCX and the DLLs it requires as described in the DLLREQ.HTM file.

Figure 10 An ActiveX control in a container

Any container would interact with the control in a similar manner.

3.3.2 ActiveX controls and the Internet

Since an ActiveX control is embeddable in a web page, Microsoft pushed to have the WWW consortium modify HTML to include embeddable objects. So now Internet Explorer can download objects over the web and install them. The main two problems with this approach are:

1. Portability
2. Security

Most ActiveX controls are compiled to run on Windows 95 or Windows NT. Therefore, they can only run on the "Wintel" platforms. This can be a problem in a heterogeneous environment.

As for security, there are many issues here which will be covered in Section 7.4, Security and Encryption. Briefly, an ActiveX control can do anything on your computer that any normal program can do. This can make for very powerful controls. However, it is impossible to verify whether or not an ActivåX control does something malicious; this is a major security hole..

Microsoft and Verisign have introduced Authenticode technology which helps certify who wrote the control. But it does not verify what the control does, and attributing a control to its author doesn't necessarily mean anything. There is an ActiveX control called "Exploder" described at *http://www.halcyon.com/mclain/ActiveX* to demonstrate the security problems with ActiveX. What the control does is wait 10 seconds, then shutdown the user's computer. Microsoft has threatened the author of the ActiveX control with legal action, so he took it off his web site.

Downloading ActiveX control randomly off the Internet is probably not the safest thing to do. The problem is not that trojan horses and viruses can exist in ActiveX controls; they can exist anywhere, including Netscape plug-ins. The difference is that ActiveX controls are so *easy* to download, install and run.

However, ActiveX controls in an intranet situation is a practical option. It is safe to assume that the servers are trustworthy and would not serve unsafe controls.

4. Java Solutions

One of the hottest topics that the media debate is: "ActiveX or Java?" As much as Microsoft's competitors like to make users think, ActiveX controls and Java applets do not necessarily compete. Java is an object-oriented language, not an object-oriented infrastructure for distributed computing. In fact, Java can make use of ActiveX controls and features of OLE (although the tools and technology that provide this are not as mature as those for C++).

Java is very similar to C++. However, one important factor is that it takes the burden of memory management away from the programmer. Memory management problems account for a large percentage of C++ bugs, so Java's built-in garbage collection is a boon to those writing large programs.

Another much-touted aspect of Java is its inherent portability. Even though languages are standardized to ease portability, very few large programs can compile on more than one platform. In addition, Java compiles into a standard bytecode, an executable format that platforms must either interpret or compile. This portable bytecode has spawned a class of programs called applets.

4.1 Applets

Even though one can write full-fledged applications in Java like any other language, one unique aspect of Java is its portable executable format. A web page can contain a Java applet that will execute when the user loads the page. This allows a web page to contain transparent active content; the user does not need to close his web browser, reboot his computer, or even tell the browser to download the program.

```
import java.awt.*;

public class Hello extends java.applet.Applet {
      public void paint(Graphics g) {
            g.drawString("Hello World!", 100, 100);
      }
}
```

Figure 11 Hello, World! applet

4.1.1 Untrusted Applets

One important difference between an applet and an application is the security necessary with an applet. A user downloads an applet without necessarily knowing that he is downloading anything at all, and runs it. Anyone could have written that applet; it does not necessarily have to be the owner of the web page. There is a potential for malice.

Therefore, untrusted applets run in a *sandbox*, a term indicating an environment that they can not break out of. In general, untrusted applets can not:

1. Read or write from persistent storage on the local machine.
2. Communicate with any computers other than the machine it was downloaded from.

Java's security model is not perfect, however. There were many security problems which, bit by bit, were fixed. As of now, there is no mathematical model of Java, which making it impossible to prove its security. The current way of testing the security is to run the implementation of Java on test cases.

4.1.2 Trusted Applets

Trusted applets are applets which do not have to run in a sandbox. Trusted applets can effectively be treated like applications.

The question is: what's the difference between a trusted applet and an application, then? Trusted applets, like untrusted applets, can be downloaded off the web. However, a trusted applet need to be extracted from a package that was signed with a digital signature. Microsoft uses a packaging format called a cabinet file. Microsoft has used cabinet files (with the extension .CAB) to distribute software for a long time. CAB files are well integrated with the Windows 95 and NT 4.0 operating systems. Sun has been pushing a type of packaging called a Java Archive (.JAR) file. JAR readers and writers are written in Java, thus ensuring their portability.

Either way, JAR and CAB files perform essentially the same function: package and compress many files in one archive, as well as hold digital signature information. It is this digital signature information that separates a trusted applet from an untrusted one.

4.2 Java Distributed Object Model (RMI)

Sun designed a distributed object model for use with Java. It uses a lightweight method invocation paradigm, called Remote Method Invocation (RMI). Distributed objects are different from local objects in that:

1. Clients of remote objects only interact with interfaces, not classes.
2. Nonremote data are passed by copy rather than by reference. The reason is that object references are only useful in the same virtual machine.
3. Remote objects are passed by reference, not by copying the actual remote implementation.
4. Since remote objects require more failure modes than local objects, each method of a remote interface must be able to throw an exception indicating this.

Java RMI is a lightweight method invocation tool to be used with the Java distributed object model. It is composed of three parts:

1. The Stub/Skeleton Layer

   The stub/skeleton layer provides the static client stubs and server skeletons. This is what the Java program will use to communicate with the lower levels.

2. The Remote Reference Layer

   This layer handles the object references and management.

3. The Transport Layer

   This layer is simply the Internet transport protocol used. Sun provides and implementation that uses TCP/IP; but someone else could replace it with UDP if he wishes.

Each of these three parts is interchangeable; they do not depend on the other layers. This allows someone to provide an alternate layer without requiring replacing the others. These layers are illustrated in Figure 12.

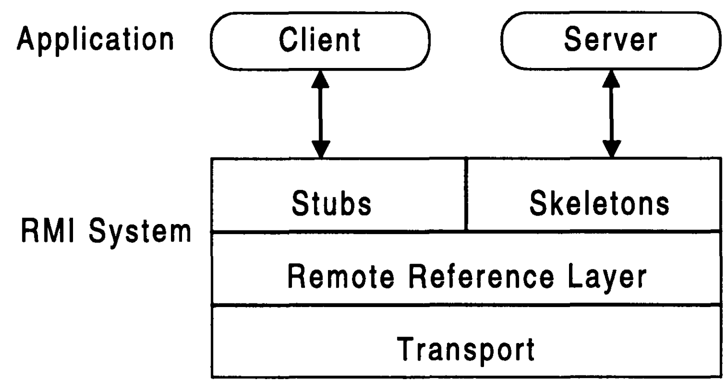


Figure 12 The Java RMI architecture [JavaRMI p.17]

4.2.1 Dynamic Class Loading

Since Java executables are the same no matter what platform the virtual machine is running in, it is possible to do an applet-like download of client stub code. This means that a Java RMI client can work with remote objects it never knew existed. The mechanism is different from that of loading an applet from a URL; the mechanism requires an intimate knowledge of the workings of Java's ClassLoader class and security.

4.2.2 ByteCode Passing

Since Java is inherently supposed to be portable, it is possible to pass behavior, not just interfaces, around. This means that method invocations can work with actual object instantiations and their bytecodes. This is one advantage that DCOM and CORBA can never have, because they involve heterogeneous networks (and they do not allow passing of executable code in parameter lists).

4.3 JavaBeans

As of recently, the only popular way of inserting downloadable components was to use ActiveX controls. While it is true that there are other embeddable object technologies in existence, like Apple/IBM's OpenDoc, web browsers do not support downloading them on the fly like with ActiveX controls.

However, Sun has recently developed a competitor to ActiveX controls, and popular web browsers will or already do support them. The infrastructure is called *JavaBeans.*

> "A Java Bean is a reusable software component that can be manipulated visually in a builder tool."
>
> [JavaBean p.9]

JavaBeans is very similar to ActiveX. Each component is called a *Bean*, and the infrastructure is designed to work well with other existing component infrastructures. There are plans to develop bridges for ActiveX, OpenDoc, and LiveConnect. This interoperability could be a boon to component software developers; they could then use mature tools like Visual Basic to design platform-independent Beans.

Java Beans are different from applets in that they enable reusability very easily. A development tool or component container can *introspect* a Java Bean to see what interfaces it supports. Java Beans are also visual, so programmers can use RAD tools to work with a Bean.

4.3.1 JavaBeans Features

Java Beans support many features of ActiveX, otherwise a good bridge could not be written. Java Beans may have a user interface (most would), and support persistence, events, and properties. Java Beans, like ActiveX controls and OLE Automation servers, are active at design time as well as runtime, to determine what properties and methods it supports. Sun calls this feature *introspection.*

With introspection, a client does not need to statically link stubs to work with an object. Like OLE Automation, a client can discover what properties and methods a Bean supports. This dynamic bindability of this component architecture is useful both at runtime and development time.

Sun is pushing "Write Once, Run Anywhere, Re-Use Everywhere" as JavaBeans' motto. The reusability part is one very important feature of JavaBeans. Through introspection, a developer can use a Bean in his own applications. The visual aspect of a Java Bean is useful in developing component software; that is one advantage a Bean has over a class library. However, objects which do not need visual modification or are immutable, such as a mathematical function library, probably do not need the overhead of Beans.

4.3.2 Bridging

With bridging JavaBeans to other infrastructures, a Bean is locked into the platforms that support those technologies. Sun is therefore promoting its "100% Java" campaign; if all components and software were written in Java, then there would be no portability problems.

The main disadvantage with JavaBeans right now is that they are an immature technology. Although Sun has had experience with downloadable objects (and correcting security problems in the sandbox model) in their Java applet technology, JavaBeans itself is immature. There are few development tools out there that will directly develop a Bean. Even though bridges will be developed, using a bridge limits a Bean to the functionality of the architecture it is bridged from.

4.3.3 Downloading and Packaging

Beans can be packaged in a compressed archive (JAR file) that contains all the classes that it needs to operate. Similar to ActiveX, a developer can sign a Bean with a digital signature that would authenticate who wrote it. This runs into the same security problems that ActiveX encounters. However, since the extent of damage is limited to the Java virtual machine, it can do less than an ActiveX control (which isn't to say it isn't much). Java Beans and ActiveX controls may both delete files on one's hard drive, but only an

ActiveX control may edit system parameters, format hard drives, or reboot a user's computer.

A client may still use Java Beans without risk by restricting himself to only using untrusted applets (and thus, the security measures are in place when the Bean is activated).

5. Specification Comparisons

Standard Objects (compiled stubs)	Dynamically Bindable Objects	Downloadable Objects
CORBA	ActiveX Controls[1]	ActiveX Controls
DCOM	Automation Servers	Java Objects (as applets)
Remote Java Objects	CORBA (DII)	JavaBeans
	JavaBeans	
	Remote Java Objects[2]	

[1] ActiveX Controls do not necessarily have to be scriptable, but since they can also be built use OLE Automation, they may be scriptable.

[2] Client stub code may be dynamically downloaded and executed (See Section 4.2.1, Dynamic Class Loading)

Table 4 Different types of objects

Since there are so many different architectures and different uses of them, any potential decision-maker needs to do the analogue of the following: "Compare apples to apples, and oranges to oranges, and ask ourselves, should we eat an apple or an orange?" In light of the number of technologies to look at, details what needs to be examined.

First there are the *Standard Objects*, where interfaces to objects are statically compiled into clients. CORBA, DCOM, and Java RMI are in this category. CORBA and DCOM are the "heavyweights", chock full of services and features. Java RMI is simply a mechanism for Java programs to communicate with each other remotely.

The second category of object technologies are *Dynamically Bindable Objects*. Clients can bind to these objects and manipulate them without knowledge of the interfaces and functions necessary; everything is done dynamically. Usually, a user will write a script that the client will interpret and use to control the component. The most popular scripting language for component software is Microsoft Visual Basic.

Finally, the third category of objects are *Downloadable Objects*. These objects are distributed in that they are components that a user downloads to his local machine and executes them. Although Java Applets and JavaBeans are not orthogonal, they are different features of the Java programming language that need to be looked at separately.

ActiveX controls are good for component software. Combined with a distributed object infrastructure like CORBA or DCOM, ActiveX controls can aid in distributing programs. JavaBeans are very similar to ActiveX controls, and they can also be combined with a distributed object infrastructure like CORBA or Java RMI, but not DCOM. Therefore, it makes no sense to compare CORBA to ActiveX controls or JavaBeans; it is possible for a vendor to mix and match if necessary.

5.1 Architectures

As shown in Table 4, each object infrastructure technology must be compared to its competitors. The main thing to remember about all these technologies is that DCOM was designed and developed to work with desktop systems originally, and CORBA was designed from a network centric point of view.

5.1.1 Standard Objects

The main heavyhitters in this arena are CORBA and DCOM. Java RMI was designed to be a lightweight protocol for use in small projects where complex management of objects was is necessary. One main area of added complexity in Java RMI which might limit it to small projects is the fact that there is no client transparency. The interfaces for working with remote and local objects are different. CORBA and DCOM abstract object location from the client, which is why they are the infrastructures designed with large object networks in mind.

5.1.1.1 Interfaces

Interfaces are the glue between clients and servers. Instead of merely publishing objects for others to use, a developer designs interfaces and/or implements them. The interfaces are then reusable by others. As published in the C++ FAQ-Lite:

> **[22.1] What's the big deal of separating interface from implementation?**
>
> Interfaces are a company's most valuable resources. Designing an interface takes longer than whipping together a concrete class which fulfills that interface. Furthermore interfaces require the time of more expensive people.
>
> Since interfaces are so valuable, they should be protected from being tarnished by data structures and other implementation artifacts. Thus you should separate interface from implementation.
>
> [Cline]

Compatible interfaces are the key to component software. A client therefore would not need to know anything about the implementation of the object; it just knows the contract that the object has agreed to implement.

5.1.1.1.1 Multiple versus Single interfaces

Microsoft's architecture to allow multiple interfaces is a good design choice; it allows an object's functionality to be shared by different interfaces. For example, suppose there are two existing interfaces (developed by two different developers): *IDictionary* and *ILookup*, as shown in Figure 13.

These two interfaces have overlapping functionality. Suppose a third vendor wishes to support these two interfaces. With DCOM, the vendor can create one object, and implement these interfaces in that object. If multiple interfaces were not supported, then the vendor would have to implement two objects. If a client wanted to use interfaces, it requests it via *QueryInterface* (in C++) or via a cast in Java.

```
interface IDictionary : IUnknown {

      // LookupWord: Looks up a word
      HRESULT LookupWord([in, string] szWord, [out, string] szDef);

      // SpellCheck: Checks the word for spelling
      HRESULT SpellCheck([in, string] szWord, [out] fBool);
};

interface ILookup: IUnknown {
      // Definition: Looks up a word's definition
      HRESULT Definition([in, string] szWord, [out, string] szDef);
      // Antonym: Looks up a word's antonym
      HRESULT Antonym([in, string] szWord, [out, string] szAnt);
};
```

Figure 13 *IDictionary* and *ILookup* interfaces

CORBA does not allow multiple interfaces yet. The OMG is currently in the process of designing a specification that allows this feature. Java objects can support multiple interfaces, which is why COM maps well into Java classes.

5.1.1.1.2 Binary and IDL Specifications

DCOM uses a binary approach for interoperability between clients and servers. Its use of C++ vtables (in the Visual C++ format) leads to the problem that it is much more complex to use DCOM in other programming languages. There are many tools for COM programming that ease the burden on the programmer at the cost of flexibility. Visual J++ is an example (see Section 5.2.2.2).

However, if the developer works with the development tool the way it is supposed to be used, then there should be no problems of interoperability between COM clients and servers.

5.1.1.2 Object Persistence

CORBA was designed with the idea of persistent objects in mind. Each object lives on a server that only exits when the server decides to kill it. COM was designed with the idea that clients create new instances of a class, and that instance is discarded when the client is done with it. The state may be stored persistently, but the actual object instantiation is transient. Both models make sense for different aspects of a distributed object system. The technologies are flexible enough so that it is possible to have COM servers of persistent objects and CORBA servers of transient objects.

5.1.1.2.1 Reference Counts

In DCOM, if the developer uses C++ to implement the client, he must keep track of reference counts. This means that every time he copies and object or stops referring to one, he needs to call *AddRef()* and *Release()*, respectively. CORBA has a similar mechanism for reference counting; the methods are *_duplicate()* and *_release()*.

The difference between these two methods is that the CORBA reference counts only count the references that are on the client itself. The DCOM reference counts work with the server itself. If a DCOM client is erroneous in its reference counting, it will affect the server and any shared objects. However, a CORBA client can not have such an effect on the server. Such a difference should not provide any argument for one technology or the other; DCOM provides keep-alive pings to determine whether a client still lives or not.

5.1.1.2.2 Object Activation

In CORBA, it is possible for the ORB to set the activation mode of the server. For instance, with the Basic Object Adapter, it is possible to set server activation policies. This allows for different methods of activation depending on the configuration:

1. *Shared server policy:* A server activated by the BOA encompassing multiple active objects.
2. *Persistent server policy:* Like the shared server except that the server is activated outside of the BOA and registered in an installation procedure.
3. *Unshared server policy:* Only one object of a given implementation at a time can be active on a server.
4. *Server-per-method policy:* The BOA starts a separate server for each method invocation; the server fulfills the request and then terminates.

With DCOM, it is impossible to do this; it always follows a shared server policy.

5.1.2 Dynamically Bindable Objects

Dynamically bindable objects refer to the class of objects that clients can access without compiling in stubs. Developers do not have to change CORBA and Java Beans to make them dynamically bindable. However, COM objects can not be made dynamically bindable; clients *must* use compiled stubs. If a client statically compiles in the *IDispatch* interface, it can then use this interface to dynamically access the methods and type information of an OLE Automation Server or ActiveX control.

For a CORBA object, the implementation does not need to do anything special. CORBA specifies that the ORB must support DII, so a client can dynamically bind to an object without having the server know whether it was dynamically or statically bound.

CORBA is more flexible than COM in that its standard objects can also be dynamically bound; nothing special has to be done on the server side. COM objects would have to provide dual interfaces, a dispinterface and a COM interface, to be dynamically bindable and efficient as a statically bound object.

OLE Automation and CORBA objects can support named arguments, optional arguments, and most ideas that are expected in powerful macro or scripting languages. Java applets and Java beans, however, are not so easily bound; the client side must download client stub code to work with remote interfaces it knows nothing about. There is a definite performance lag in downloading client stub code and running it. Basically, the best thing to do in Java working requiring dynamically bindable objects is to use a Java ORB and forget Java RMI.

5.1.3 Downloadable Objects

With the advent of the Internet and simple and automatic downloading of objects, two main players have arisen: ActiveX controls and Java applets. Java Beans are new and not widely used, but Sun is pushing its use very strongly.

Downloadable objects differ greatly from the previous two in that method invocations are not remote; all objects are downloaded from a remote source to the local system. This paradigm is a combination of:

1. Downloading the component.
2. Automatically installing the component.
3. Letting component containers on the local system know that there is a new component they can embed.

Java Beans and Java applets can run in a sandbox, so security would not be a problem. If desired, they can run as trusted applets, much like ActiveX controls. Java applets are simply downloadable objects. ActiveX controls and Java Beans are downloadable components.

5.1.3.1 Components vs. Applets

Component technologies like ActiveX and JavaBeans enable software reuse. Each component can provide a single unit of functionality that can work with others. While there are other component technologies in existence like OpenDoc, their main disadvantage is that the infrastructure to automatically download and use them is not in place.

Applets were the forefathers of downloadable objects on the WWW. However, they lacked many features which ActiveX controls and Java Beans have; namely, they do not lend well to reuse. Without the source to an applet, using it in one's own applications is very difficult. ActiveX controls and Java Beans are similar to class libraries; they allow a vendor to provide functionality without releasing source code.

5.1.3.2 Components vs. Class Libraries

Components are more modular than class libraries. Instead of spending efforts worrying about library conflicts and compatibility problems (which can end up being nontrivial), a software developer can simply use standard interfaces to work with each component. Class libraries need to be compiled for each different compiler since the epilog/prolog code of each compiler is different.

That is not to say that class libraries are never useful. Often an object does not need a visual interface, even for customization. In this case, it is often unnecessary to add in the overhead to make it a component. These libraries often do not need to be used in RAD tools or over the Internet as active content. In these cases, class libraries are a good, efficient choice.

5.2 Languages and Development Tools

Language	Technology				
	DCOM	**ActiveX Controls**	**OLE Automation**	**CORBA**	**Beans/Applets/RMI**
Ada95	✓			✓	
C	✓	✓	✓	✓	
C++	✓	✓	✓	✓	
COBOL[1]	✓			✓	
Eiffel[2]	✓		✓	✓	
Java[1,2]	✓	✓	✓	✓	✓
Pascal[3]	✓	✓	✓		
Smalltalk	✓	✓	✓	✓	
Visual Basic[4]	✓	✓	✓		

[1] COBOL and Java IDL mappings are in the process of being specified
[2] Current Eiffel and Java IDL mappings are not official OMG mappings (yet); they are vendor specific
[3] Borland provides Pascal support from their Delphi development environment
[4] Visual Basic - Controls in VB5.0

Table 5 Native language support of various distributed object technologies

In selecting a technology to work with, the user needs to decide leverage knowledge that he already possesses. Table 5 displays the technologies mentioned in this thesis and what languages natively support them. Using object technology bridges and/or mixed-language development tools, a programmer can program for any of these technologies using almost any language. The main problem with this is that the application will take a performance hit.

The next sections examine a few languages or development tools for each technology. Note that the language coverage is not exhaustive of all possibilities; this section simply compares two languages per technology. At the end of each section, there is a subsection talking about the pros and cons of each language, along with a flow chart to help make a decision. Please remember that the flowchart is not exhaustive, and every project will have different aspects that can not be reflected in the flowchart.

5.2.1 CORBA Language Support

CORBA is a specification, not a particular implementation of an object technology. The ORB vendors must support an IDL mapping to a language to fully support it. The mapping allows servers and clients to talk to each other through the ORB. To communicate with objects developed with other ORBs, both ORBs must implement IIOP, now a requirement to be called CORBA-compliant.

In an impatient rush to provide support for certain languages, some vendors have implemented their own IDL mappings. The problem with this is that once a user uses a proprietary IDL mapping, it could be difficult to move the program to another ORB when the OMG provides an official mapping.

5.2.1.1 C++ Programming

The IDL data types map to C++ data types. Since IDL is a subset of C++, C++ programmers can leverage their knowledge to write IDL quite easily. Table 6 is a list of the primitive data type mappings.

IDL Type	CORBA Mapping	C++ Definition
short	CORBA::Short	short
long	CORBA::Long	**Platform Dependent**
unsigned short	CORBA::Ushort	unsigned short
unsigned long	CORBA::Ulong	unsigned long
float	CORBA::Float	float
double	CORBA::Double	double
char	CORBA::Char	char
boolean	CORBA::Boolean	unsigned char
octet	CORBA::Octet	unsigned char
longlong	CORBA::Longlong	**Platform Dependent**
ulonglong	CORBA::Ulonglong	**Platform Dependent**

Table 6 IDL C++ mappings of primitive data types [VisiCPPPG 11-2]

All interfaces map into C++ pure abstract base classes, which the client and server share. In fact, using abstract base classes to represent an interface to an object is often a good programming technique; it enhances readability and maintainability [Cline]. IDL Inheritance maps to public inheritance in C++. The client also gets stub code generated, and the server gets skeleton code generated. To implement the server, the programmer has to derive a class from the server skeleton code and implement the interfaces.

An interface may also work with other objects. The catch with this is that since objects are only passed by reference in CORBA, any object that a user wishes to pass around *must* be defined in an interface, which may not be possible for legacy code. As a result, the OMG is currently in the process of specifying a *pass-by-value* specification where an entire object and its contents can be passed to a method. Writing a pass-by-value specification is more complex than "packing up the bits and sending them"; custom marshaling is required, something which the spec will address. IONA technologies has already implemented a proprietary version of this feature in their Orbix ORB.

Since C++ is not a type-safe language, programmers often like to pass around pointers to buffers (the Win32 API has many good examples of this). IDL does not provide any method to handle this buffer passing. Even if a user wants to pass an array, the array must be defined to be a fixed lenght. To simplify this, the OMG has specified a sequence template which encapsulates an array with a length. This is used in lieu of buffers.

As for programming in C++ itself, code written for the client is more interchangeable between different ORBs than the server. The reason is that the client has had two rounds of standards definitions; the server has only had one. Soon however, server implementation will be made more portable.

5.2.1.1.1 DII Client Language Issues

As for languages to make DII requests in, the best choice is "neither". DII is best used as an underlying mechanism of a higher level language. So far, there are no scripting languages (at least on the Windows platform) that work with DII natively. The best thing to do is to use one of the Automation/CORBA bridges supplied by ORB vendors and work with Visual Basic. Each DII request in C++ or Java will have to be tediously constructed and then requested. It just adds complexity to the whole program, which is what component software is trying to eliminate.

5.2.1.1.2 Client

For a client, the program must initialize the ORB and then bind it to an object with something similar to the following code:

```
CORBA::ORB_var orb = CORBA::ORB_init();
// MyObject is an abstract base class
MyObject *myobject = MyObject::bind();
```

There should be a try/catch block around the bind invocation to catch any exceptions. Basically, however, this is all that is needed to obtain a pointer to an object. To invoke methods, one can simply make standard C++ method invocations like

```
myobject->Foo();
```

A client is, in general, intuitive to implement with C++.

5.2.1.1.3 Server

For a server, the program must provide an implementation for the interface. The implementation must inherit from skeleton code produced by the IDL compiler. For an interface `MyObject`, a skeleton class `_sk_MyObject` is created. The programmer must then inherit from this skeleton class:

```
class MyObject_impl : public _sk_MyObject {
      // Method Declarations
};
```

Once the implementation is done, the programmer needs to write a server that can interact with the ORB and register the object. A program must initialize the ORB, use the ORB to obtain an object adapter, and use the object adapter to let the ORB know that there is an object available for remote users to use. Some sample code follows:

```
// Initialize ORB.
CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);

// Obtain the Basic Object Adapter from the ORB
// To read about the BOA, see Section 2.2.3
CORBA::BOA_ptr boa = orb->BOA_init(argc, argv);

// Create the object
MyObject_impl myob();

// Let the object adapter know the object is ready to
// receive requests
boa->obj_is_ready(&myob);

// All the objects have been initialized; enter a loop
boa->impl_is_ready();
```

That is all it takes in C++ to get a CORBA object running.

5.2.1.2 Java Programming

Since Java is very similar to C++, so are the IDL mappings. Also, since the primitive data types in Java take the same amount of memory no matter what platform the program is running on. Therefore in Java, there is no need to prepend each primitive data type with a "CORBA" namespace or class like in C++.

Unlike C++, however, there is no OMG endorsed Java mapping. All the current IDL mappings are proprietary; there are subtle difference between them. Current implemented mappings include those from Visigenic, IONA, and Sun. It is probably best to wait for the OMG endorsed mapping before writing much code in Java. However, ORB vendors who currently provide Java IDL mappings map them almost exactly the way that C++ does; the difference becomes syntactic, not semantic.

5.2.1.3 CORBA Language Decision

If one were to implement and use CORBA objects, the interaction between CORBA and the language should really not be a factor, since programming in C++ or Java is very similar. The project manager should decide which language to use based on which language is best suited for the task.

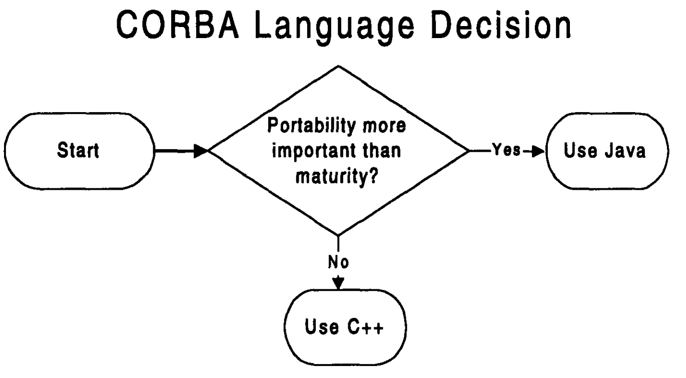


Figure 14 CORBA Language Decision

Java is more portable than C++ and it is type-safe; there is also no need for manual allocation and deletion of objects. Java is the only language that allows downloadable, sandboxed applets. Netscape Communications is including Visigenic's Java ORB runtime in the newest version of their web browser, Navigator 4.0. The distribution of an ORB with a major web browser will open the doors to a wave of widely distributed n-tiered client/server systems.

The main problem with Java is that it is not very mature. C++ is years ahead of it in terms of legacy code, class libraries, and development tools. The choice of language depends mainly on the type of project and its goals.

5.2.2 DCOM Language Support

Unlike CORBA, DCOM is a binary standard. DCOM specifies the wire protocol and interactions between machines and objects. In addition, COM objects must support arrays of pointers in a specific format to work. In Visual C++, a C++ object using virtual functions will automatically have its virtual table in the proper format to work with COM; other compilers must be modified to work with COM as transparently.

```
interface ISequentialStream : IUnknown
{
  [local]
  HRESULT Read(
     [out, size_is(cb), length_is(*pcbRead)]
           void *pv,
     [in] ULONG cb,
     [out] ULONG *pcbRead);

  // Extra methods removed
}
```

Figure 15 Excerpt of a Microsoft MIDL file

Therefore, a user must check to make sure a compiler will work with COM objects, even if the Table 5 lists that it does. For example, Borland C++ and Visual J++ will work with COM, GNU's g++ and Sun's Java Developer's Kit will not.

As mentioned in Section 3.1.3, MIDL and RPC, one can write interfaces in MIDL if necessary. MIDL can help when the object requires information to marshal its arguments. Figure 15 is excerpt from a Microsoft IDL file. Figure 16 is a description of the parameters.

Note that MIDL, unlike OMG's IDL, lets an interface to contain a variable-sized buffer or array (*void *pv* in our example). When a pointer is passed between two processes, the data it referenced is invalid in the destination process. When MIDL generates proxy code, it automatically generates marshaling code that will copy the referenced data. In the case of an array, the size of the buffer passed in is specified by the *size_is(cb)* tag, and the length of the data that is returned is specified by the *length_is(*pcbRead)* tag. The marshaling code generated by MIDL will therefore look at the tags to ensure that the appropriate number of bytes are passed when the function returns. MIDL is based on DCE's RPC's IDL; it is more complex than IDL, but it is more flexible, too.

void * *pv,*	Pointer to buffer into which the stream is read
ULONG *cb,*	Specifies the number of bytes to read
ULONG * *pcbRead*	Pointer to location that contains actual number of bytes read

Figure 16 Description of *ISequentialStream::Read()*'s parameters

5.2.2.1 C++ Programming

A COM object is assigned a GUID called a CLSID; this unique identifier represents the class. Each CLSID is associated with a group of GUIDs called IIDs; each IID represents a COM interface that the object implements.

COM interfaces, much like CORBA IDL interfaces, map well to C++ abstract base classes. Each method in the interface must return an HRESULT type parameter to indicate success or failure; any return values essential to the logic of the method must be passed as an out parameter, like *ISequentialStream::Read()*'s *pcbRead.*

In the examples that follow, no error checking code is included, although they should be used in production-quality code.

5.2.2.1.1 Client

A client can not access a COM object directly; it must only access an object through its interfaces. An interface is represented by an abstract base class. Such an object can never be instantiated; instead it merely acts as a template to access interfaces of the object.

Suppose in our example below, MyClass implements COM interfaces *IUnknown* and *IMyClass*, and *IMyClass* contains one method, *MyMethod()*. Here is one way to obtain an interface that can be used later:

```
IUnknown *pUnk;
CoCreateInstance(CLSID_MyClass,      // The CLSID of the object
                 NULL,               // Pointer to aggregating object
                 CLSCTX_SERVER,      // Context of object implementation
                 IID_IUnknown,       // IID to request
                 &pUnk);             // Address of pointer to interface
```

The client must first declare a variable to hold a pointer to an interface. This example uses the *IUnknown* interface since all COM objects must implement it. The client then calls *CoCreateInstance* with the appropriate parameters. The NULL parameter is used in aggregation, the way that COM implements reusability. This is covered in Section 5.3, Reusability. The CLSCTX_SERVER parameter is a constant which indicates the type of server the client would run, whether it is instantiated in the client process, out of the client process on the same machine, or on a separate machine.

After the function call, *pUnk* will contain a pointer to a valid interface. The client can use this interface to get pointers to other interfaces, if it wants:

```
IMyClass *pMyClass;
pUnk->QueryInterface(IID_IMyClass, &pMyClass);
```

That is all that is necessary to obtain an interface pointer. If there were no error, then the client can call *pMyClass->MyMethod()* as if it were a normal C++ method invocation. When the client is done using the object, he must decrement the reference count. *QueryInterface* automatically gives an object a reference count of 1. If the client obtains more interfaces to the object, it will need to call *AddRef* to increment the reference count. When the client is done with each reference of the object (including the first reference obtained by *QueryInterface*), it must then call *Release* on that reference.

5.2.2.1.2 Server

Similar to CORBA servers, COM servers require implementation of the object as well as the actual connections that allows the operating system to access it. Implementing an object to implement the MyClass COM object, the programmer must declare the following class:

```
// Implementation of IMyObject
```

```
class CImplMyObject : public IMyObject{
public:
	// Constructor
	CImplMyObject();

	// IUnknown members
	STDMETHODIMP QueryInterface(REFIID, PPVOID);
	STDMETHODIMP_(DWORD) AddRef();
	STDMETHODIMP_(DWORD) Release();

	// IMyObject members
	STDMETHODIMP MyMethod();

	// Reference Count
	int m_cRef;
};
```

STDMETHODIMP is simply a macro that returns HRESULT (*typedef'*d to a 32-bit integer) and other modifiers. The *STDMETHODIMP_(*type*)* is a similar macro that returns *type* instead of HRESULT. Below are implementations of the methods.

```
CImplMyObject::CImplMyObject(){
	m_cRef = 0;
}

HRESULT CImplMyObject::QueryInterface(REFIID refiid, PPVOID ppvoid){
	if (riid == IID_IUnknown || riid == IID_IMyClass){
		*ppvoid = this;
		return NOERROR;	// Constant indicating no error
	}

	return E_NOINTERFACE;	// Constant indicating the interface
				// does not exist
}

DWORD CImplMyObject::AddRef(){
	return ++m_cRef;
}

DWORD CImplMyObject::Release(){
	if (--m_cRef == 0)
		delete this;
}

HRESULT CImplMyObject::MyMethod(){
	cout << "MyMethod invoked" << endl;
	return NOERROR;	// constant indicating success
}
```

This was a very simplistic example, it was a class that implements one interface. When a class implements more than one interface, however, the number of lines necessary to implement the object, as well as the complexity, increases tremendously. Not only must the programmer define a new class for each implementation of an interface, but there must also be a class that implements *IUnknown* as well as keeps references to instances of the other classes. The programmer must implement quite a bit of "grunge code", code that is necessary for housekeeping purposes.

As described in Section 5.3, Reusability, COM objects do not support direct inheritance of binary code. Rather, they require a user to *implement* aggregation. This is just a layer of complexity which C++ programmers need to deal with. Java deals with reusability in a more intuitive way.

The previous sections described how an object is implemented. More work is needed to link it to the COM architecture and operating system registry so that a client can invoke it. A COM server can be written as one of three things: an in-process DLL, a local server (yet a standalone executable), or a remote server. The main difference between a local server and remote server is that the remote server is on a machine separate from the client, so a programmer can talk about them as if they were one. The difference between an in-process DLL and a server is that the DLL is loaded in the process of the client, and the server is in a separate process. An in-process DLL can call methods without resorting to copying data and context-switching. However, if an in-process DLL crashes, then the whole program crashes. If a server crashes, the client can keep on running, if it was designed to do so.

The object must be compiled into a binary, either a DLL or executable. Once it is placed on the system, the operating system must be told where to find it. In Windows 95 and Windows NT, this is done through registry values. In the case of a remote server, the registry would have an entry that tells on what machine the object is on.

To work with the COM architecture, the server or DLL must expose an object that exposes COM interface *IClassFactory*, declared in Figure 17. A class factory is an object that will create instances of a COM object. Each class factory is separate from the actual object it creates.

```
interface IClassFactory : IUnknown {
   HRESULT CreateInstance(IUnknown *pUnkOuter,
                          REFIID riid,
                          void **ppv);
   HRESULT LockServer(BOOL fLock);
};
```

Figure 17 The *IClassFactory* interface [Brockschmidt2 p.232]

Whenever a client calls *IClassFactory::CreateInstance*, the class factory will create the object and the client can use it from there. The reason why there were no class factories in the client example above is that *CoCreateInstance*, used in the example, automatically creates a class factory and uses it. A class factory is useful to keep around if the client wants to efficiently create many instances of the same class.

There are many other aspects in implementing COM servers. To get more detailed information, one of the best references is *Inside OLE* by Kraig Brockschmidt [Brockschmidt2].

5.2.2.2 Java Programming

The idea of Java working with COM objects may seem antithetical to Sun's philosophy in creating Java. After all, Java is a portable language, and COM (as of now) primarily exists on Windows platforms (see Section 7.1, Platforms). However, Microsoft has licensed

Java from Sun, and in their Visual J++ compiler, they have extended the Java bytecode specification to work with COM objects.

Since Java is a type-safe language, people tend to put Java at a higher level of abstraction than C++. Instead of making Java equivalents of class factories, *CoCreateInstance*, and other low-level details, Microsoft designed Java-COM interaction to be as intuitive as possible. Thus there are major differences in programming COM between C++ and Java.

5.2.2.2.1 Client

Visual J++ can only use COM objects which have type libraries associated with it. The reason is that in its effort to integrate COM with Java, Microsoft converts Java type libraries into Java interfaces and objects using the Java Type Library Wizard. The bytecode files generated use Microsoft extensions that interface to the COM architecture below. These extensions do not have any Java language equivalents; thus they must be generated with the extension.

Assuming that all that has been done, working with an object is rather simple. Suppose these is a COM object *MyClass*, which implements two interfaces, *Interface1* and *Interface2*. The client needs to work with the class through its interfaces; working with the class directly is legal in Java, but if the class is a COM object, then there will be a runtime error.

Thus:

```
Interface1 myInterface = (Interface1) new MyClass();
```

will execute correctly, but

```
MyClass myClass = new MyClass();
```

will cause a runtime error.

This is equivalent to C++'s *CoCreateInstance*. Once the client has the interface, it can work with it like any other Java interface. To work with another interface of the same object, the client has to cast it:

```
Interface2 myInterface2 = (Interface2) myInterface;
```

This is an intuitive way of working with COM objects instead of working with *CoCreateInstance*, *QueryInterface*, and reference counting. Everything is automatically handled by the Java subsystem. Java's use of multiple interfaces used by an object almost mirrors COM's paradigm.

It is important to note that using Java to instantiate COM objects is less flexible than using it C++. *CoCreateInstance* provides a lot of options (such as server location) that Java's *new* operator cannot possible integrate and maintain the syntactic integrity of the Java language. Thus, if the client needs to make heavy use of COM instantiation (thus optimally using a class factory) or wants to make use of different servers, Java is probably not the best language to use.

5.2.2.2.2 Server

To implement a COM server in Java, the programmer must create the Java class, and then register it as a COM object in the registry. Microsoft provides a program, *javareg*, that will do this for the user.

There are just a few rules that the programmer must follow in writing a COM object. Whereas in C++, each method of an interface must return HRESULT to indicate success or failure, in Java, each method must return *void* and throw an exception to indicate failure.

The Microsoft Java virtual machine knows how to automatically aggregate an object, which lets programmers inherit normally with the *extends* keyword. All Java-created COM objects are aggregatable, but not all COM objects written in C++ are. Also, since Java does not support multiple implementation inheritance, Java COM objects can not aggregate from more than one object. These are the only limitations that Java programmers are faced in reusability with COM.

For more details on development with COM clients and servers, the reader should look in the Visual J++ help files.

5.2.2.3 DCOM Language Decision

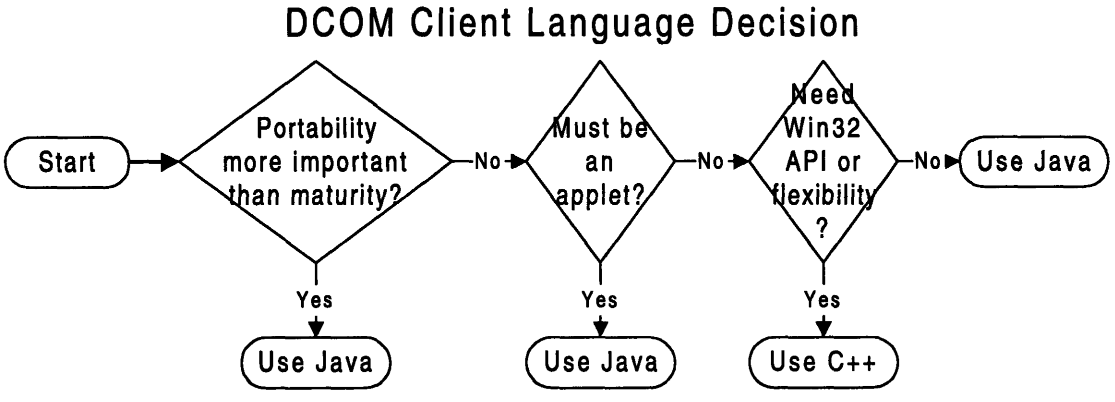

DCOM Client Language Decision
Start
Portability more important than maturity?
No
Must be an applet?
No
Need Win32 API or flexibility ?
No
Use Java
Yes
Use Java
Yes
Use Java
Yes
Use C++

Figure 18 DCOM Client Language Decision

Writing a COM client in Java is more intuitive to Java programmers than writing a COM client in C++ to C++ programmers. In Java, the programmer does not have to worry about reference counts, class factories, or *IUnknown* and its members. Microsoft's adaptation of the Java language to COM simplifies removes the low-level grunge with little compromise of functionality.

An untrusted Java applet can not access COM objects; the only types of Java programs that may do so are trusted applets and applications. In addition, the virtual machine must be one that will support Microsoft's Java bytecode extensions. The only Java virtual machine which supports this is Microsoft's own Java VM.

One of the major disadvantages of COM programming in C++ is the client side maintenance of reference counts. It just adds an extra aspect of complexity; code shared by many programmers can misuse reference counts; this can cause objects not to be deleted when the client overestimates the reference count, or even worse, it may cause a mission-critical program to crash when it underestimates the reference count and deletes an already-deleted object.

However, it may not be a good move to jump to Java just yet. Keep in mind, while making your language choice, that the Java tools to work with COM are still maturing.

On the server side, it is probably best to work with C++ for now. C++ integrates very well with the Win32 API and is not restricted in functionality like Java is (for portability's sake). The reference counting problem is not an issue with servers, since they do not actually count references; they just provide functionality for clients to do so. Note, however that C++ servers need to explicitly aggregate for object reuse; Java servers can just inherit. Also, C++ servers must be written to allow aggregation, Java servers are automatically aggregatable. Java, in this case, removes a lot of grunge code. However, Java servers can not inherit from more than one parent. This is not necessarily a problem, but this means that Java can only use one other COM object efficiently; every other reused object must be contained (see Section 5.3, Reusability).

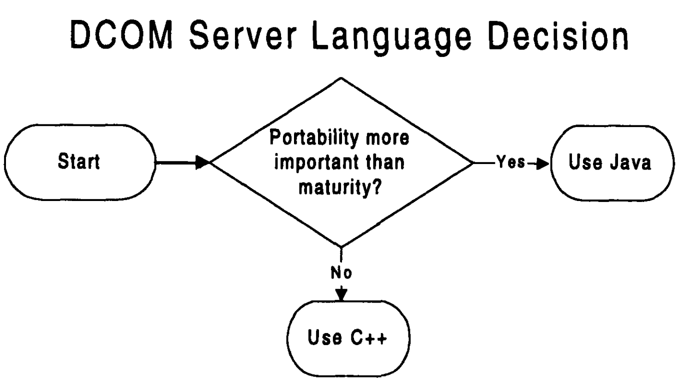

DCOM Server Language Decision
Start
Portability more important than maturity?
Yes
Use Java
No
Use C++

Figure 19 DCOM Server Language Decision

Although in this section a distinction is made between client and server, a network of objects is not necessarily going to be a simple client/server n-tiered hierarchy. As systems become more complex, each executable will contain clients and servers of different types of objects. Thus the above statements are just guidelines for project managers to use.

5.2.3 OLE Automation

Method	Description
Invoke	Given a dispID and other necessary parameters, calls a method or access a property in this dispinterface.
GetIDsOfNames	Converts text names of peroperties and methods to their corresponding dispIDs.
GetTypeInfoCount	Determines whether there is type information available for this dispinterface, returning 0 (unavailable) or 1 (available).
GetTypeInfo	Retrieves the type information for this dispinterface if GetTypeInfoCount returned successfully.

Table 7 The methods of ***IDispatch*** [Brockshmidt2 643]

Even though OLE Automation uses COM does not mean that the same languages used for COM development are ideal for working with this higher level technology. This section examines writing OLE Automation servers in C++ and Visual Basic. OLE Automation is not a very effective or secure way to work with distributed objects. Remote Automation (Section 3.2.3) is included only for completeness, and Microsoft seems to have plans to phase out its use. The distributed path of OLE Automation is probably to work with *IDispatch* objects via DCOM.

5.2.3.1 C++ Programming

Writing an Automation client in C++ is probably not the best thing to do. To invoke a method from a dispinterface, a C++ client would have to go through a lot of low-level grunge to construct the request, and then call *Invoke()*. It is best to leave dispinterface method invocations up to a high level language.

Programming an OLE Automation server in C++ is a matter of implementing the *IDispatch* interface, as mentioned in Section 3.2.1. Once the dispinterface is designed, the programmer must implement the *IDispatch* interface so that all clients can invoke methods of the dispinterface. From here, the programmer can create a type library using MIDL to facilitate use of the object in programming environments; type libraries aid in error detection before the program is actually executed.

When an object implements a dispinterface through its *IDispatch* methods and also implements a COM interface that shares the same methods as the dispinterface, it is said to have a dual interface. This lets clients call the object's methods as if it were an Automation client or as a COM client. It increases the adaptability of the object so its clients would not have to work with a type of interface just to work with the object.

5.2.3.2 Visual Basic Programming

Microsoft Visual Basic is a very popular language. According to Deloitte and Touche Consulting Group, 40% of CIOs rate Visual Basic the most important to their technology plans [CSVB]. It is not surprising, considering that Visual Basic is a development tool that allows application development with little code writing.

OLE Automation arose from Microsoft's Visual Basic team. They needed a method for their language to dynamically bind and work with objects. Therefore, it is no surprise that Visual Basic integrates very well with OLE Automation.

Visual Basic is better than C++ when it comes to writing Automation clients. With Visual Basic, one does not need to worry about constructing an invocation every time he wants to use a dispinterface method. This is why Visual Basic is used very often for front end work; combining Automation servers and controls together is perfect for this development environment.

5.2.3.3 OLE Automation Language Decision

OLE Automation was designed for dynamic binding and scripting. Dynamically binding to an object simply adds a lot of unnecessary grunge code (and therefore, a source of bugs) to a program. So unless a client absolutely *must* use C++ for its work, it should best be implemented in Visual Basic or some other Rapid Application Development tool.

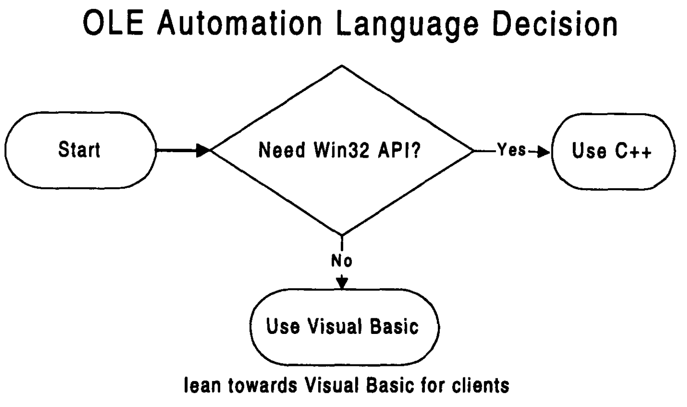

OLE Automation Language Decision
Start
Need Win32 API?
Yes
Use C++
No
Use Visual Basic
lean towards Visual Basic for clients

Figure 20 OLE Automation Language Decision

For the server, it does not matter as much. If it will only be used by Automation clients, then Visual Basic should be fine, if it can do that job. If COM clients need to use it too, then the server should be implemented in C++ with dual interfaces for flexibility.

5.2.4 ActiveX Controls

As of now, Microsoft has not specifically set down what the minimum interfaces are that an ActiveX control must implement. Until recently, user could only create ActiveX controls with Visual C++. However, many different programs can *use* ActiveX controls. Many ActiveX controls can also be controlled through OLE Automation. This is very powerful for active documents (hence the name)

5.2.4.1 Rapid Application Development Tools

As of this writing, only two RAD tools support ActiveX control development: Visual Basic 5.0 and Delphi 2.0. Visual Basic 5.0's Control Creation Edition is currently in beta testing, so there are not many experiences on developing controls with it yet. Visual Basic, however, has always been effecting in developing containers that use ActiveX controls.

Borland's Delphi has been a very powerful RAD tool for a long time, based on its time-tested Turbo Pascal engine. Delphi works very similarly to Visual Basic, containing ActiveX controls and manipulating them, too. Delphi can also create reusable Visual Component Libraries (VCLs) that other Delphi projects can reuse. These VCLs are where one can create ActiveX controls from. Apiary Software, a developer of component software tools, has a utility called OCX Expert that converts most VCLs into OCXs seamlessly and easily. This utility makes Delphi a strong contender as a development tool of ActiveX controls.

The main problem with RAD Tools is the size of the produced executables. ActiveX controls that are installed with software can be any size, but components that are downloaded on the fly on a need basis need to be small in size, at least to cater to the many home users running 28.8k modems.

5.2.4.2 C++ Tools

Most ActiveX controls on web pages are written in C++. Even with Visual C++, there are many ways to develop controls. Each have their advantages and disadvantages.

5.2.4.2.1 Straight C++

It is possible to create an ActiveX control simply by creating a COM object that implements all the necessary interfaces. Most ActiveX controls use the same implementations of these interfaces, so constantly reimplementing these interfaces in C++ is very tedious. Straight C++ might be a good way to understand how an ActiveX Control works, but it is probably not something that use every time would want to do every time.

5.2.4.2.2 ActiveX Template Library

Recognizing that many ActiveX controls use similar implementations of certain interfaces, Microsoft released the ActiveX Template Library. The ATL can help in creating COM objects, Automation servers, and ActiveX controls. The ATL still has bugs (also because of the immaturity of C++ compilers' template compiling routines), and has a high learning curve. However, ActiveX controls created with the ATL have a very small footprint.

5.2.4.2.3 Microsoft Foundation Classes

In Visual C++, a user can use the ContolWizard to create an ActiveX control. Creating a control using the ControlWizard is a lot simpler than creating one with straight C++ or the ATL. However, these controls require a lot of memory for two reasons:

1. Since controls created this way use the Microsoft Foundation Classes, these controls require the MFC DLLs on the client's machine. If the client does not already have them, then the files must be downloaded and installed.
2. The ControlWizard generates and uses lots of excessive code which is not necessarily used. Therefore the file size of the controls are generally large. A release version of an ActiveX control that does nothing is 22k, compiled with Visual C++ 4.2b for the Win32 platform.

5.2.4.3 ActiveX Control Language Decision

In writing an ActiveX control container, it basically depends on how powerful the container should be. There is a tradeoff here between ease of development and power of the tool. A discussion of the benefits and tradeoffs of RAD versus C++ programming is beyond the scope of this thesis.

If ActiveX controls are to be downloaded over the Internet, especially via modem, then they should be small and lightweight. If ActiveX controls are installed via an intranet or "sneakernet", then there wouldn't be a problem with size, barring client machine memory considerations.

In general, controls written in Visual Basic or Delphi will be large. Controls written with the ControlWizard are smaller, but there is still code bloat. Controls written with the ActiveX Template Library will be very small. Even though controls written in straight C++ will be very small, the complexity involved in creating a useful control this way outweighs the benefits in speed. For Internet use, controls should be written with the ATL or ControlWizard; time will tell to see if future versions of Visual Basic or Delphi will create small controls.

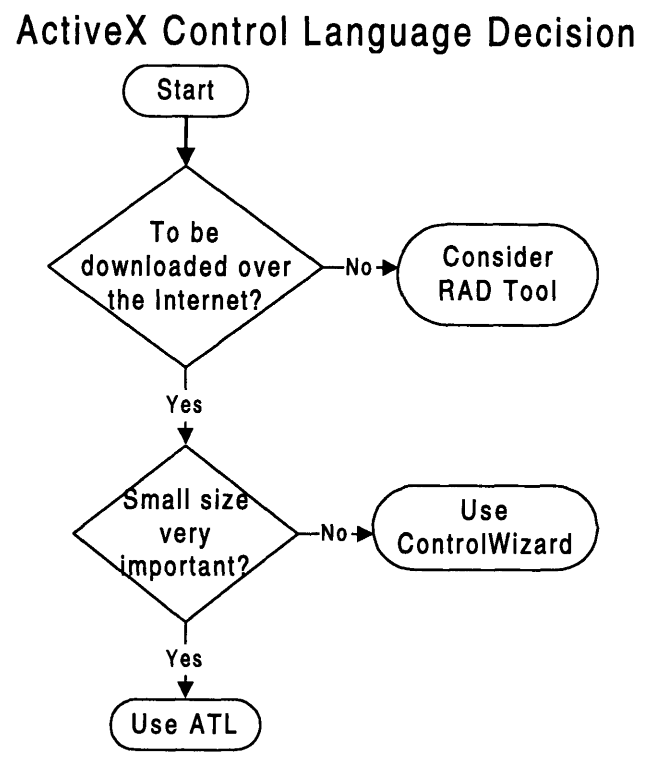

ActiveX Control Language Decision
Start
To be downloaded over the Internet?
No
Consider RAD Tool
Yes
Small size very important?
No
Use ControlWizard
Yes
Use ATL

Figure 21 ActiveX Control Language Decision

5.3 Reusability

One major problem that object oriented technologies are supposed to solve are in the area of software reusability. Hardware engineers have known this idea for a long time; designing a microprocessor is very complex. To solve these complexities hardware engineers reuse designs.

Software has had a limited form of reuse in terms of class libraries for a long time. When object oriented languages became popular, objects were designed to enable reusability through class inheritance. At a language level this was good, but at the linking and compiling level, it was not always so simple. Libraries had to be recompiled to run on different compilers and operating systems. Legacy systems had to be ported from mainframes to run on client/server systems. Such conversions contribute little to the functionality of the system and take up much time and money.

The object oriented technologies presented enable reusability at many different levels. "Reusability" is a general term that can be interpreted many ways. This section examines:

1. Interface reusability
2. Implementation reusability
3. Legacy systems integration

5.3.1 Interface Reusability

"Interface reusability" refers to the ability of one interface to build from another interface. This would provide for clean IDL files whose class hierarchies can be shared and reused by other developers.

Both MIDL and IDL allow interface reuse through interface inheritance. This allows an interface hierarchy to be constructed which other developers can use; instead of implementing a bloated interface, they can simply implement an interface higher up on the hierarchy. Plus, since interfaces are not language-specific, different interfaces in the same hierarchy can be implemented in different languages.

5.3.2 Implementation Reusability

Technology	Implementation Reuse Methods		
	Inheritance	Containment	Aggregation
DCOM	Java Only	Yes	Yes
ActiveX Controls		Yes	
OLE Automation		Yes	
CORBA (Stub and DII)	Yes	Yes	
Java Objects	Yes	Yes	

Table 8 Implementation Reusability Methods

The idea of implementation reusability is an important aspect that object oriented technologies address. Implementation reusability is the idea of reusing executable code without needing the source. The object oriented technologies in Table 4 all allow some form of reuse. There are many ways to reuse code, and each object oriented technology uses one or some of them.

5.3.2.1 Inheritance

Inheritance is the most common way to reuse code in object oriented languages. If an object oriented technology mirrors a language's reuse capabilities. CORBA and Java based objects use inheritance for implementation reusability. COM objects are restricted to either containment or aggregation for reusability. However, Microsoft has developed their Java virtual machine to allow COM object reuse through inheritance. In other words, they abstract the rote implementation of aggregation away from the user. Inheritance is a basic object oriented method; it is described in Section 1.2.3, Inheritance.

5.3.2.2 Containment

One method of reuse that is rarely used by non COM programmers is *containment.* Suppose object A wishes to use methods in object B. Object A can thus contain an instance of object B and invoke methods on it as necessary. Object A is thus the containing object and object B is the contained object. Object A can thus call object B from it's own members; there can be a one-to-one correspondence between object A's methods and object B's.

Containment allows for great flexibility on what is exposed by the containing object. With containment, the containing object may hide some members or methods of the contained object. This is impossible with inheritance or aggregation.

Figure 22 is an example of containment in C++. Note that the implementations of these objects are in the header file, something that should not be done to work with these component infrastructures. The *AddDollars* method is hidden in the containing object, and the *AddNT* method calls it after converting the NT currency into Taiwanese currency.

However, this extra flexibility comes with a price; the *ShowBalance* method in *CNTAccount* calls the *ShowBalance* method in *CDollarAccount.* This is just an extra level of function calling. Plus, duplicating all the methods in a large contained objects is prone to error and adds complexity.

Since interfaces of containing objects inherit from the interfaces of contained objects, there is often no need to hide methods of a contained object. Often, if a method of a contained object needs to be hidden in the interface of a containing object, the blame should be placed on poor design of the contained object's interface.

```
// contain.h
// Example of containment

// Bank account saved in US currency
class CDollarAccount{
public:
   CDollarAccount(){
      m_nMoney = 0;
   }

   void AddDollars(double nDollars){
      m_nMoney += nDollars;
   }

   double ShowBalance(){
      return m_nMoney;
   }
private:
   double m_nMoney;
};

// Bank account in Taiwanese currency
class CNTAccount{
public:
   void AddNT(double nNT){
      m_Account.AddDollars(nNT / 27);
   }

   double ShowBalance(){
      return m_Account.ShowBalance();
   }
private:
   CDollarAccount m_Account;
};
```

Figure 22 Containment example

Microsoft discourages containment in COM, and encourages aggregation. All the other technologies can use containment if they truly wish to, but it is not the sole vehicle for reuse. In summary, containment should not be used for well thought out interface hierarchies, and should be used only when necessary.

5.3.2.3 Aggregation

Aggregation only works when an object model can supports multiple interfaces. Since Microsoft eschewed inheritance in COM it advocates aggregation as the method of reusability in COM.

In COM (in C++), when a client wants to obtain an interface pointer from an object, it needs to call *QueryInterface*. Since the object itself implements *QueryInterface*, it can return an interface pointer from anywhere.

Suppose there are the interfaces as described in Figure 23. Note that these interfaces are different from those in Figure 8; the *IPolice_Car* interface does not inherit from *ICar*. Thus, these two interfaces have nothing to do with each other.

```
import "unknwn.idl";

[uuid(00000000-0000-0000-0000-000000000000),
      object]
interface ICar : IUnknown {

      HRESULT GetSpeed([out] float *speed);
      HRESULT SetSpeed([in] float speed);
      HRESULT GetDirection([out] float *dir);
      HRESULT SetDirection([in] float dir);
      HRESULT drive([in] float distance);
};

[uuid(00000000-0000-0000-0000-000000000001),
      object]
interface IPolice_Car: IUnknown {

      HRESULT GetSiren([out] BOOLEAN *on);
      HRESULT SetSiren([in] BOOLEAN on);
};
```

Figure 23 Interfaces in aggregation example

Now if let us implement two objects, *CCar* and *CPolice_Car*. *CCar* implements all the methods of *ICar*. *CPolice_Car* implements both methods of *IPolice_Car*, and also contains an instance of *CCar*. Also, *QueryInterface* is also implemented so that whenever a client asks for *ICar* it will return a *ICar* pointer to the *inner* object. That is aggregation.

Figure 24 explains the paradigm in pictures. The client sees *CPolice_Car* exposing the *ICar* and *IPolice_Car* interfaces. However, inside, *CPolice_Car* doesn't implement *ICar*; it only merely implemented its *QueryInterface* so that the interfaces are exposed in this manner.

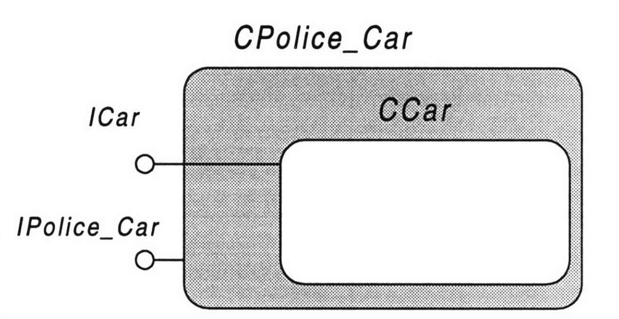


Figure 24 Aggregation model

COM objects must worry about a few things to allow aggregation. First of all, COM objects are not automatically aggregatable; suppose someone obtained *ICar* from *CPolice_Car* above, then called *QueryInterface* looking for *IPolice_Car*. *CCar*'s *QueryInterface* must know how to delegate to its outer object *CPolice_Car*. Plus there are things a containing object must worry about concerning the lifetime of a contained object, such as controlling the reference count, worrying about reentrancy in its destructor, and other things. Errors in implementing aggregational functionality can cause subtle bugs in an object concerning the management of its inner object. Especially in a distributed object environment, these bugs can be hard to track down.

Many COM programmers, especially beginning ones, would prefer not to do aggregation, so they sometimes implement reusability through containment. Microsoft has recognized this problem. When one writes COM objects in Java, they are automatically aggregatable, and aggregation is abstracted away from the programmer through inheritance. Microsoft could do this with Java since they modified the Java virtual machine; there would have been too many obstacles towards modifying their C++ compiler to do the same thing.

5.3.2.4 Component Object Reuse

ActiveX controls have a mechanism for reusing Windows standard controls, like the Edit controls called *subclassing*. However, this does not work when one wants to reuse third party components.

The best way to reuse a third-party ActiveX control your own component is to make your component an ActiveX control that also acts as an ActiveX container. This way, it is possible to reuse a contained control's features at the programmer's discretion.

5.3.3 Legacy Systems Integration

Most companies have a significant investment in mainframe systems or some technology that existed before these object technologies. Some of these systems are programmed in RPG, COBOL, FORTRAN, or old database systems. The time it takes to convert these resources for integration into a modern network is often not worth the time and money, so companies might be stuck with inefficient distributed systems. However, there are options to integrate older systems into modern object networks.

5.3.3.1 ORB Available on Legacy System

There are many ORB vendors out there. Chances are that any modern system will have an ORB that runs on it. If a legacy application exists on one of these systems, making use of this is obvious. Parts of a legacy application can be coarsely wrapped by an object; the object would delegate function calls to the legacy application. If the legacy application provided all its functionality through a command line (like many Unix applications) then the delegation would involve forking a new process and running a command line. Otherwise, hopefully the program is modularized enough for an object to delegate to API calls. Figure 25 illustrates this.

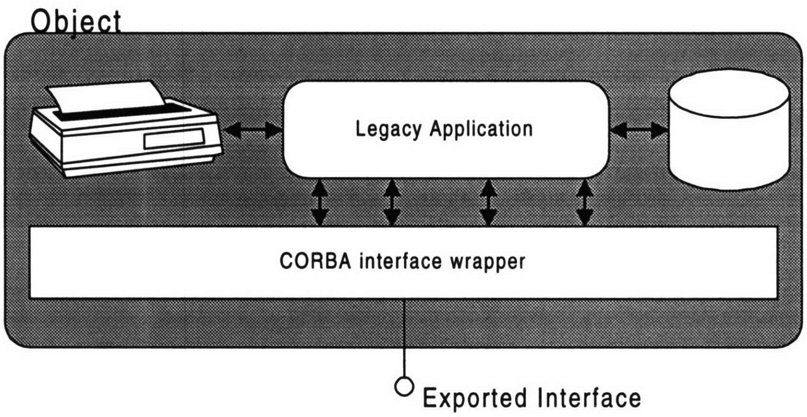


Figure 25 Legacy application with an ORB wrapper

5.3.3.2 ORB Unavailable on Legacy System

If there is no available ORB on the system and no vendor is willing to port to one, there are two options:

1. Write an IIOP wrapper
2. Wrap it with an application on an ORB available platform

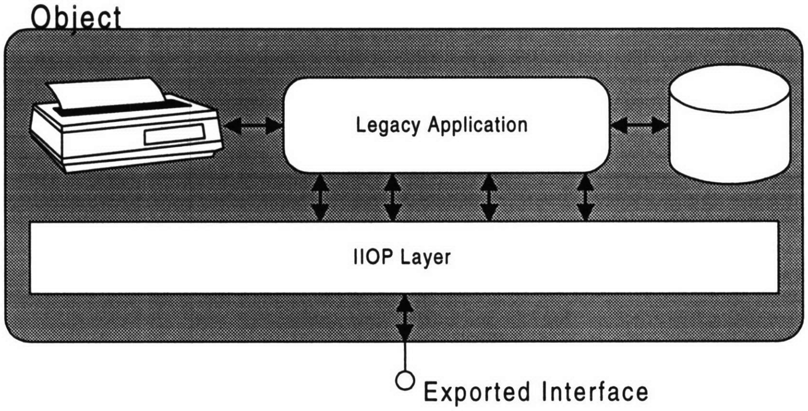


Figure 26 IIOP wrapped legacy application

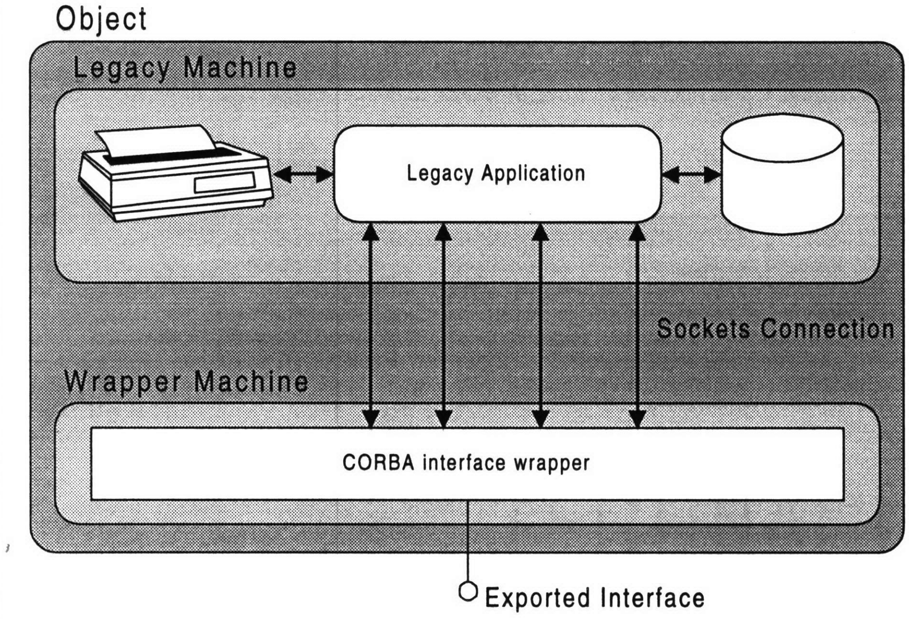


Figure 27 Legacy application with a remote wrapper

If the legacy application does not need much of the services of an ORB, then a developer may construct an IIOP wrapper. IIOP is a well-defined protocol using TCP/IP that any vendor can work with. Thus, any IIOP-compliant ORB may work with the legacy application. This is illustrated in Figure 26.

Writing an IIOP wrapper may not always be trivial; handling things like invalid objects and other requests are just administrative details that have nothing to do with the legacy application's functionality. It may then be possible to write a client/server system where the client will communicate with the legacy application via sockets. The client can then have an ORB installed on it that will communicate with the other objects on the system. This is illustrated in Figure 27.

Note that in all cases, invoking objects do not see the workings of the object; it need not even know that the object is a legacy application. All this is abstracted by the interface.

5.4 Services

All these infrastructures do not mean much if it is solely an infrastructure. A superhighway system, for instance, would mean nothing if there were buildings in the area to generate commerce that will actually use it. Thus the two major infrastructures, CORBA and DCOM, provide a set of services that clients can use in their programs. The services range from object management and manipulation to business logic to vertical market integration. The main point is that each of these services are accessed through standard interfaces; all CORBA vendors must follow the interfaces in implementing the services, as well as all DCOM vendors (when non-Windows DCOM support becomes widely used). This section compares the services provided by each of these infrastructures; note that in the case of CORBA, these are only specifications, and not all CORBA vendors have implemented all of them.

Note that this section does not talk about security; this issue is dealt with in Section 7.4, Security and Encryption.

5.4.1 Naming and Object Location

CORBA, as mentioned previously, was developed from a network-centric point of view. With this in mind, the OMG specified a Naming and Trader service that can be used to build a comprehensive directory service. The Naming service provides a hierarchical naming structure that can be used with many existing directory services (*e.g.* DCE, LDAP, etc.). An object can get bound with a name in different contexts; this is similar to a file system's directory traversal paradigm.

The Trader service allows a client to look up objects grouped by function and interfaces as opposed to by name. The Trader service is to the Naming service as the Yellow Pages are to the White Pages. Optimally, a client would use the Trader service to look up objects, and then dynamically bind to them.

DCOM has no Trader or Naming service. Specifying one should not be hard for Microsoft to do, especially since the OMG has already done it for CORBA. However, COM objects cannot dynamically bind to objects so a Trader service would provide less functionality for DCOM than CORBA. OLE Automation objects can be dynamically bound, and ActiveX controls can be downloaded and installed on the fly. A Trader service would work best with these technologies.

5.4.2 Relationships and Lifecycles

CORBA objects can be connected with relationships. The Relationship service provides a graph of related objects that can be traversed by a client. Two of the most common relationships are Containment and Reference relationships. Their meanings should be obvious from their names.

The Lifecycle service allows copying, removing, and moving objects. A Lifecycle service implementation that works in conjunction with the Relationship service relieves a programmer from having to worrying about deep copies, moves, and deletions.

DCOM, without something similar to a Lifecycle or Relationship service, must let the programmer manually handle object references and implement changes to contained or referred objects himself. This is prone to error, and in potentially complicated object graphs, this can be a source of bugs.

5.4.3 Events

In CORBA, an object can listen into an event channel. This object is called an event *consumer*. The object that generates the event is called the *supplier*. The event channel decouples the communications between the supplier and consumer; many consumers can listen to a particular event channel and many suppliers can write to one. Events are useful for asynchronous notifications of information. In CORBA, the supplier and consumers do not need to know each other.

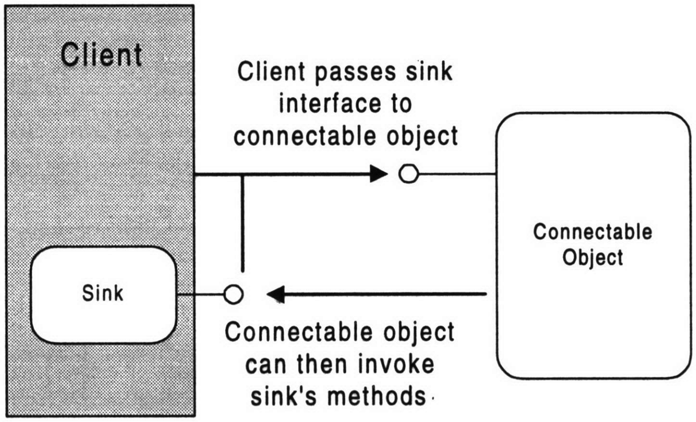

Client
Client passes sink interface to connectable object
Connectable Object
Sink
Connectable object can then invoke sink's methods

Figure 28 Connectable Object

The closest thing to events in the DCOM world is an infrastructure called *Connectable Objects*. It is a mixture between CORBA events and object reference passing. In this model, there are three entities: a client, a connectable object, and a sink. Without getting into technical details (such as what interfaces are used, etc.), what happens it that the client passes the sink's interface pointer to the connectable object. The connectable object can then invoke operations on the sink's pointer as it needs to. There does not need to be a bijection between connectable objects and sinks; surjections and injections are possible.

Connectable objects are more similar to simple CORBA object reference passing than to events. However, the specifications require for enumerability of a connectable object's sinks. This also allows for asynchronous notification of events. Using Connectable Objects as a means of asynchronous event notification requires more on the part of the programmer than using the CORBA Event service.

5.4.4 Transactions

A transaction is an interaction in which there are two possible outcomes: success and abort. If there is an error in the transaction, any effects of the transaction up to that point are rolled back to the state of the system before the transaction started.

CORBA has provided a powerful Transaction service. It supposed flat and nested transactions, transactions can span heterogenous ORBs, and it is easily integratable into existing IDL interfaces.

COM's support of transactions is limited to the transactions in the structured storage interfaces for stream I/O. The current Windows NT and Windows 95 implementations of structured storage also do not support this functionality (in the specification, transactioning is an option). So even though the official Microsoft specification of transactioning is limited to stream writes, no operating system has implemented it. Clearly, in the case of transactions, DCOM does not face up to it.

5.4.5 Persistence

An important part of what these services need is a consistent interface for persistence. Persistence is the idea of "hands off storage." The client does not have to worry about the details of saving an object, and sometimes the object's state is always saved after each change.

In CORBA, there are the persistence and externalization interfaces. The name of the persistence service is a set of interfaces that make up the Persistent Object Service

(POS). The POS can accommodate a number of storage devices such as SQL databases, object databases, file systems, and compound documents. The POS defines three layers of abstraction that hide different implementations.

1. Persistent Objects: Objects whose states are persistently stored.
2. Persistent Object Manager: Implementation independent interface for persistence operations. It provides a uniform view of persistence in the system across multiple data services.
3. Persistent Data Services: Interfaces to the particular datastore implementation.

Depending on the level of control the client needs, the persistence functionalities may be used at any of these levels.

On the DCOM end, the persistence interfaces provide a consistent method of letting an object provide persistence capabilities. However, it only provides one layer of abstraction; it lacks the granularity of CORBA's POS.

5.4.6 Querying and Database Access

CORBA also has interfaces that deal with querying and databases. The Query service provides consistent interfaces to work with databases and extract objects from them (the objects may actually be in the database, or they may just represent data in the database). CORBA also specifies a Collection service which provides interfaces that aid in grouping objects and working with them; this is useful for working with results of queries.

Microsoft has had a lot of practical experience working with consistent ways to access databases. The Windows operating systems have had Open Database Connectivity (ODBC) for a long time; it is a time-tested method of consistently accessing information from different database sources. They have recently designed a set of interfaces called OLE DB that will let applications consistently access data from more data sources, including ODBC compliant sources. Not only can one query data from databases, but also from file systems, other COM components, spreadsheets, etc. ODBC currently only allows SQL queries; OLE DB allows much more.

It is hard to say which service is better; there are few applications that make use of either of these technologies. As of now, Microsoft has the advantage of its experience in designing the ODBC API.

5.4.7 Licensing

COM and CORBA both support licensing of objects with consistent interfaces. For COM, they specify an interface, *IClassFactory2*, which extends the *IClassFactory* interface. The *IClassFactory2* system differentiates between two scenarios:

1. The machine or end user is fully licensed.
2. The machine or end user is not licensed at all.

It is up to the object implementer to define the method of authentication. It can be as simple as a text file on the hard disk to a complex Kerberos protocol.

The OMG has also defined licensing interfaces, but they provide more functionality than COM's. The licensing scheme can be more granular; it is possible for a server to know when the component is being used, keep track of how many method invocations are called, etc.

5.4.8 Versioning

One issue that component vendors must face is that of versioning. When an object provides new functionality in the next version, how should it be reflected, and still provide compatibility with the old version? The OMG is in the process of specifying a method of consistent versioning.

Microsoft ignores this issue by saying that if a user wants to implement more functionality, they need to create a completely new interface. The new functionality is available through the new interface, and the object can also implement the old interface too, so older clients can use it. This is an instance where multiple object interfaces fits in nicely.

6. CORBA Implementations

CORBA Implementations (I)

General					
Organization	**BEA**	**Visigenic**	**IBM**	**Sun**	**HP**
Product	**ObjectBroker**	**VisiBroker**	**SOM**	**Joe/NEO**	**Orb Plus**
Development License	Unavailable	$2995-4995	Free	195	$3,000
Runtime License	Unavailable	$150-250	Free	195	$100
Free Evaluation		Yes	Yes		Yes
Capabilities					
IIOP	Yes	Yes	Yes	Yes	Yes
IR	Yes	Yes	Yes	Yes	Yes
Static	Yes	Yes	Yes	Yes	Yes
Dynamic	Yes	Yes	Yes	Yes	Yes
COM Gateway	Yes			Yes	Yes
Language Bindings					
C	Yes		Yes	Yes	Yes
C++	Yes	Yes	Yes	Yes	Yes
Java	Yes	Yes	Yes	Yes	Yes
Smalltalk	Yes	Yes	Yes		Yes
Cobol		1998	Yes		
Ada					
Services					
Naming	Yes	Yes	Yes	Yes	Yes
Events	Yes	Yes	Yes	Yes	Yes
Life Cycle	1998	Yes	Yes	Yes	Yes
Trader	Yes	Yes	Yes		Yes
Transactions	Yes	Yes	Yes	Yes	1998
Concurrency	1998	Yes	Yes	Yes	1998
Security	Yes	Yes	Yes	Yes	Yes
Persistence	1998		Yes	Yes	1998
Externalization	1998		Yes		1998
Query	1998	1998	1998	Yes	
Collections	1998	1998	1998		
Relationships	1998		Yes	Yes	
Time	1998	Yes	1998		
Licensing	1998		1998		
Properties	1998		1998	Yes	

Figure 29 CORBA Implementations (PART 1)

CORBA Implementations (II)

General					
Organization **Product**	**IONA** **Orbix**	**Expersoft** **PowerBroker**	**ICL** **DAIS**	**Chorus** **CoolORB**	**U. Berlin** **JacORB**
Development License	$2500-7500	$2995-5900	$2995-4995	$1000-4500	Free
Runtime License	$100-200	$50-100	$50-100	$5-95	Free
Free Evaluation	Yes		Yes	Yes	Yes
Capabilities					
IIOP	Yes	Yes	Yes	Yes	Yes
IR	Yes	Yes	Yes	Yes	
Static	Yes	Yes	Yes	Yes	Yes
Dynamic	Yes	Yes	Yes	Yes	
COM Gateway	Yes	Yes			
Language Bindings					
C	Yes		Yes		
C++	Yes	Yes	Yes	Yes	
Java	Yes	Yes	Yes		Yes
Smalltalk	Yes	Yes			
Cobol	Yes		Yes		
Ada	Yes				
Services					
Naming	Yes	Yes	Yes	Yes	Yes
Events	Yes	Yes	Yes		Yes
Life Cycle	Yes	Yes	1998		
Trader	Yes	Yes	Yes		
Transactions	Yes	Yes	Yes		
Concurrency	Yes	1998			
Security	Yes	Yes	Yes		
Persistence	Yes	Yes	Yes		
Externalization	Yes	Yes			
Query	Yes	1998	1998		
Collections	Yes	1998			
Relationships	Yes	Yes			
Time	Yes	1998	1998		
Licensing	Yes	1998	1998		
Properties	Yes	Yes	1998		

Figure 30 CORBA Implementations (PART 2)

Previous sections have looked at comparisons of the specifications. However, not all implementations of CORBA actually implement all those services. Plus there are issues involved with "porting" programs from one ORB to another. Client has two rounds of standardization. CORBA clients have had two rounds of standardization, thus they are much easier to port than servers.

This section takes a look at some ORBs on the market and see what they have to offer, in terms of CORBA services and their own add-ons. The commercial-grade ORBs add extra features like fault-tolerance, replication, and extra management tools (these are not specified by the OMG). Below are some brief descriptions of popular ORBs. Tables of these ORBs can be found on pages.

6.1 BEA - ObjectBroker

Originally owned by Digital, ObjectBroker now belongs to BEA. Digital is one of those vendors in the Microsoft camp, so they decided to sell of their CORBA ORB and go with DCOM. ObjectBroker is one of the most mature ORBs, first shipping in 1991, and is running on more systems than any other ORB today.

6.2 Visigenic - VisiBroker

VisiBroker will possibly run on more desktops in the world when Netscape releases their newest browser. Its C++ version runs on Solaris, SunOS, HP-UX, and 32 bit Windows. However, its integration with Windows is not as good as it could be; having no COM gateway like some of the other vendors. This precludes its development using Windows-native RAD tools like Visual Basic.

6.3 IBM - SOM

The System Object Model (SOM) was developed by IBM, and is available on OS/2, AIX, and Windows. One example of CORBA on the desktop is OS/2 Workplace shell, where all objects are SOM objects. This shows that CORBA can be used for fine-grained desktop applications instead of just large networks.

Since IBM is trying to promote the use of SOM, it is free for download.

6.4 Sun - Joe/NEO

NEO is the name of Sun's CORBA-compliant ORB. Joe is another ORB that is implemented in Java, by Sun. Joe is compatible with NEO. Sun also lets users download a free Windows connectivity package that allows real-time integration with COM objects. The packages run on Windows and Solaris.

6.5 Hewlett-Packard - ORB Plus

Hewlett-Packard's ORB Plus is a mature ORB that integrates very well with Windows. It runs on HP's own operating system, Solaris, and Windows NT. It provides a very streamlined COM gateway. Its Windows versions also includes wizards that integrate with Visual C++ and simplify repetitive tasks.

6.6 Orbix

IONA Technologies' Orbix runs on more platforms than any of the ORBs described here. Orbix supports many CORBA services, and adds many of its own features. Its COM gateway also integrates with the Visual Basic user interface so CORBA objects can be treated as OLE Automation objects. Orbix is one of the most mature ORBs in the industry.

6.7 Expersoft - PowerBroker

Expersoft is a consulting company that also wrote the PowerBroker ORB. It is a mature ORB used widely. It runs on Windows NT, Solaris, AIX, and HP-UX. However, its COM gateway is sold separately from its ORB, so a user would have to spend more money to integrate with COM.

6.8 ICL - DAIS

ICL Technologies is an information technology consulting company that specializes in distributed object solutions. Their DAIS ORB runs the largest CORBA implementation in the world right now, on Windows and Solaris. This ORB is primarily used in the United Kingdom as in only recently coming into the United States through another company, DiaLogos. DAIS, however, does not have a COM gateway.

6.9 Chorus Systems - CoolORB

Chorus Systems' main products are real-time operating systems. They developed CoolORB to run on this operating system. Essentially, they give themselves an advantage over other RT systems in that embedded systems using their product can work with CORBA object, but can also work with other CORBA vendors via IIOP. To optimize communications when inter-ORB communications is not necessary, users may use Chorus IPC as their transport mechanism instead of IIOP.

In addition to their own operating systems, CoolORB will run on AIX, Linux, SCO operating systems, Solaris, and the 32 bit Windows OS's. Fortunately, it is possible to download their older versions to evaluate it.

6.10 University of Berlin - JacORB

JacORB was developed by Freie Universität Berlin in Germany. It is not a commercial product, but instead, an example of an ORB written completely in Java. It is published under the GNU Public License. Since it is written in Java, anyone with a JDK 1.0 can work with it.

7. Implementation Comparisons

CORBA implementations generally do not keep up with the specification as fast as DCOM. The reason CORBA does not keep up as well is that:

1. There are many companies working together on the specification; thus it is necessary for a specification to be published before actually making much leeway into an implementation.
2. DCOM is, in general, controlled by Microsoft (this may change with the passing of DCOM into the Open Group). As such, it is a lot easier to implement the specifications before they are actually published.

Much of CORBA is specified together by competitors; it's not perfect, but it was designed to cater to many tastes. DCOM was built out of Microsoft's desktop operating systems. Thus, a delay in DCOM means a delay in the next release of the operating system. DCOM therefore must face market pressures, perhaps tangential ones too.

7.1 Platforms

	CORBA	DCOM	Java (Beans, Applets)	ActiveX Controls
AIX 4.1	Yes	No	Yes	No
BSD 2.1	Yes	No	Yes	No
Digital Unix	Yes	Beta	Yes	No
HP/UX 10.x	Yes	No	Yes	No
HP/UX 9.x	Yes	No	Yes	No
IRIX 5.x	Yes	No	Yes	No
Linux	Yes	Beta	Yes	No
MacOS 7.5	Yes	No	Yes	Beta
MVS	Yes	Beta		No
Open VMS	Yes	No	Yes	No
OS/2 Warp	Yes	No	Yes	No
QNX 4.22	Yes	No		No
Solaris 2.x	Yes	Beta	Yes	No
Sun 4.1.x (Solaris 1.0.1)	Yes	No	Yes	No
VxWorks 5.1.1	Yes	No		No
VxWorks 5.2	Yes	No		No
Windows 3.1	Yes	No		No
Windows 95	Yes	Yes	Yes	Yes
Windows NT 3.5	Yes	No	Yes	Yes
Windows NT 4.0	Yes	Yes	Yes	Yes

Table 9 Platform Availability

Table 9 is a snapshot of the industry in Spring 1997. The blank entries indicate that the data is not available. There are many other platforms that support CORBA, especially Unixes and mainframes for legacy applications. Java, ActiveX controls, and DCOM are mostly restricted to modern operating systems.

Since CORBA is defined by the OMG, which is composed over 700 companies, it is no surprise that CORBA implementations abound. Note that even though all these platforms support CORBA, not all of the implementations are provided by the same vendor.

Microsoft is only provides DCOM on Windows and the Macintosh. All the other platforms (the Unix ones) that support DCOM are provided by Software AG, which has an alliance with Microsoft.

There are many vendors out there that support Java. Sun has provided development kits for Java that provide full access. For those platforms which Sun has not provided any developer's kits, Netscape's web browser allows applet access to applets. Once Netscape's browser supports JDK 1.1, these platforms can then use JavaBeans.

The fact that DCOM comes with Windows NT 4.0 makes it very popular among Windows users. Windows 95 does not include DCOM, but it is freely available off of Microsoft's web site. One major disadvantage of DCOM is that it *requires* a Windows NT Server to be running on the network. For information infrastructures based on Netware or Unix, DCOM is therefore not even an option.

DCOM's intimate integration with the operating system lets the infrastructure work hand in hand with the operating system. For instance, in COM it is possible to load objects from a DLL by running it in the local process, dynamically binding to the DLL. No CORBA implementation allows this, partially since it is tough to do in a consistent manner across different platforms.

7.2 Software Development

Developing with distributed object technologies can be more difficult than developing traditional applications. The main difference is that developing a distributed object solution, or even a simple client/server solution, debugging and testing is harder. There is not only one thread of instruction in such applications.

This section discusses the available tools to work with these technologies.

7.2.1 COM and CORBA Development

COM development can be aided by Visual C++ and the Microsoft Foundation Classes. Even without MFC, COM server and client development is mostly straightforward. This is only with C++ development. Borland C++ is also another development tool that can work with COM. C++ is the primary language used to develop COM software, so there are other C++ compilers that can be used. Microsoft's COM support is best for C++.

If Java were the language used, then the only tool available is Microsoft's Visual J++. As of version 1.0, it did not have excellent COM support; the only hints of COM support were in Microsoft's Java SDK and the Visual J++ help files. Few books have been written on the topic. In general, Java support for COM is not mature yet.

For other languages, Microsoft does not produce the software itself. There are Ada and Eiffel compilers that support COM, but since they are not as widespread as C++ or Java, chances are that it is harder to find help when using those tools.

As for CORBA, there are few if any tools out there for CORBA specific development. To require a CORBA object is rather straightforward and needs little +handholding. See Section 9, Example CORBA Implementation, for an example of an implementation using Visigenic's ORB.

Developing Java remote objects should be as simple as working with Java objects themselves. The reason is that Java remote objects does not require much (nor does it provide as much). The main problem that developers will encounter with this will probably just be in the debugging phase; Java development tools are just not as mature as those for other languages.

7.2.2 Visual Development

Outside of traditional programming, there are development tools that require visual interaction with the programmer. For ActiveX controls and Java Beans, Rapid Application Development (RAD) tools are very helpful (or necessary). However, for more low-level control or for efficiency reasons, programmers may opt to not use a RAD tool, but use a traditional tool to develop these visual objects.

RAD tools are ideal for components that:

1. Describe itself through properties, type information, or other means
2. Facilitate graphical editing of its properties
3. Are directly customizable from a programming language
4. Generate events so the RAD tool can semantically link components

7.2.2.1 OLE Automation Server Development

Visual Basic is the chief tool for OLE Automation server development. Using Visual Basic to develop software is very simple since it is visual (hence the name). To develop an OLE Automation server or client, one simply has to load the project and modify its properties and such visually. The development environment (and language) handles much of the plumbing necessary to route the appropriate messages and method calls.

7.2.2.2 ActiveX Control Development

ActiveX Control developers have many choices choices, as mentioned in Section 5.2.4. The choices include Visual Basic (whose virtues were extolled in the previous section), the ActiveX Template Library, and Visual C++'s OLE ControlWizard. It is also possible to use a very powerful RAD tool by Borland called Delphi to create ActiveX controls (although a third party program is necessary).

Visual development is the intuitive way to develop ActiveX controls. Users can use RAD tools to see what a container or component will look like and make modifications. However, one may argue that too much is abstracted away from the programmer. At its heart, ActiveX controls developed with the ControlWizard is straight C++. The ControlWizard just creates a framework of C++ code which the user fills in with necessary information. If a programmer relies on the ControlWizard too much, he won't be able to do extensive debugging of his own control, and will not learn much about the internals of ActiveX controls as he should. The tradeoff is there.

7.2.2.3 Java Bean Development

The newest in the bunch is JavaBeans. Not only is Java a new language, but JavaBeans is such a new technology that few vendors have had time to put out tools that actually introspect Beans for rapid development.

One of the most notable players in the field is Borland, coming out with its JBuilder product. JBuilder uses the RAD interface of Borland's groundbreaking Delphi. This is essentially a Visual Basic for Java, more so than Visual J++. JBuilder will not only support Java Bean development, but it will also use Visigenic's Java ORB to develop Java CORBA applications.

7.3 Business Issues

The main bone of contention between CORBA and DCOM is that CORBA was designed by a conglomerate of companies, whereas DCOM was designed solely by Microsoft. Microsoft also does not seem to be relinquishing its hold on COM, although it is trying to make DCOM's ORPC an Internet Standard. Microsoft is the main supporter of DCOM and ActiveX. It has licensed DCOM to Software AG and has given parts of ActiveX to the Open Group (formerly OSF, makers of RPC). Only time will tell whether or not other platforms will be as supported as Windows.

CORBA has been controlled by the OMG, with more than 700 members. This may imply slowness in the frequency of new specifications and vendor compliance. As more and more companies join the OMG, which is now the largest specifications group in the world, it is hard to tell if this will affect its task.

7.3.1 Price

The main disadvantage of CORBA is the price. DCOM comes free with Windows NT 4.0 Server (which is *not* free!) and the Java solutions come with Sun's free Java Development Kit. However, there are free CORBA ORBs out there. These ORBs are generally not for commercial quality software, though, and support for them may be hard to get.

Good news exists for CORBA users, though. Netscape has made a deal with Visigenic to include VisiBroker runtime software with every Netscape Navigator 4.0 and Netscape Enterprise Server 3.0. With the prevalence and popularity of Netscape's software, it is likely that a new generation of CORBA-enabled applets will arise that take advantage of Java with the power of an ORB. At the least, it is more likely to be used than DCOM because COM constructs in Java code does not comply with official Java's specified behavior, even if it runs on a platform that supports it.

On the downloadable objects playing field, all those technologies are also free. ActiveX control capability come with Windows, and as implied above, JavaBeans support is also free.

7.3.2 Support

There are many vendors betting on the success of one of these technologies. Microsoft is the main proponent of DCOM and ActiveX, Sun is busily promoting Java solutions, and the OMG and its members are pushing CORBA. Microsoft has already submitted its DCOM wire protocol ORPC to the Internet Engineering Task Force to join the grass-roots origins of the Internet. Users and developers generally would not need to worry about finding where to turn for updates, upgrades, and questions.

7.3.2.1 CORBA Support

The OMG with its members is a major driving force that will increase the popularity of CORBA; if not by technological then by sheer numbers of use. Whether it will ever overtake the huge share of personal computer running Windows operating systems (and thus will probably tend towards COM/DCOM) remains to be seen. Even if CORBA fails on the Windows platform, it is likely to succeed on non-Intel, non-Windows platforms, despite Microsoft and Software AG's efforts. The one notable exception to this is Digital, which recently sold its ObjectBroker ORB. Instead, it has embraced DCOM as the distributed object technology it will use in the future.

Besides CORBA vendors, there are numerous other vendors who will use CORBA in their upcoming products. Below are some major companies whose role in adopting CORBA will affect developers.

7.3.2.1.1 ORB Vendors

The OMG consists of 700+ members. Many of these members have developed their own ORBs which they may distribute with their products. The influential ones in the industry include Hewlett-Packard, IBM, Sun, and Xerox.

There are many other members of the OMG, who do not develop ORBs, but will support CORBA in their services and products. The sheer mass of companies supporting CORBA implies that it will not go down easily.

7.3.2.1.2 Netscape

Perhaps the most notable of all those supporting CORBA is Netscape. As mentioned above, it is planning putting Visigenic's ORB in their newest web browser and server. This will practically guarantee the longevity of CORBA unless something drastic happens.

Netscape is Microsoft's biggest competitor in the web browser market. It is one of the few markets where Microsoft is not ahead. Not surprisingly, Netscape does not want Microsoft to gain market share; DCOM support or acceptance by the Internet community can do that. To do everything it can to slow down the proliferation of DCOM, Netscape is pushing CORBA.

7.3.2.1.3 Borland

Borland, like Netscape, is one of Microsoft's competitors. Borland is Microsoft's biggest competitor in the compiler market, ever since the days of MSDOS. Therefore, Borland's JBuilder will use Visigenic's CORBA implementation as the method underlying client/server solutions.

In addition, JBuilder is one of the first RAD tools for Java Beans. While Visual J++ is having trouble working with JDK 1.1 (a requirement for Java Beans), Borland has a chance to leap ahead and popularize CORBA on the Windows platform.

7.3.2.1.4 Silicon Graphics

Silicon Graphics, manufacturers of well-known graphics processing machines, have started including IONA Technologies' Orbix ORB into their IRIX operating system. They will also include bundled application software using Orbix. Providing this in the operating system is similar to Microsoft providing DCOM with Windows NT, gibing developers easy access to object technologies. The implementation distributed with IRIX will interoperate with Orbix on other platforms.

7.3.2.1.5 Oracle

Oracle, the large database company, has decided to integrate Visigenic's CORBA implementation into its Network Computing Architecture (NCA). They will include the architecture in the NCA development environment. With Oracle targeting the WWW, providing a CORBA development environment will generate a steady flow of CORBA programmers.

7.3.2.1.6 Novell

Novell will license and distribute Visigenic's ORBs with their IntranetWare™ server platform. Considering how Novell's Netware has a large and loyal following, it is likely that this will affect many companies. The server platform will encourage the use of CORBA and Java for development of applications.

7.3.2.2 DCOM and ActiveX Support

The main supporter of DCOM and ActiveX is the manufacturer, Microsoft. Since these are the crux of Windows NT 4.0's infrastructure, and Microsoft intends to migrate all their operating systems to Windows NT in the future, Microsoft will compete all it can to make sure these technologies proliferate. To help make this happen, they are attacking their main weakness: platform unavailability (and legacy systems). To solve this, they have made alliances and have also tried to make ActiveX an open standard.

7.3.2.2.1 Digital

Digital and Microsoft have had a long-standing relationship, ever since Windows NT started running on the DEC Alpha. With DCOM being built into Windows NT, Digital had to include it in their DEC Alpha versions, and therefore are adopting DCOM. In addition, they have sold their ORB, ObjectBroker, to BEA Systems.

DCOM support on the powerful DEC Alpha machines (whether running NT or Digital Unix) will let developers use DCOM in powerful super-applications. However, this is a niche market; the main thrust of the DCOM evangelism will come from Microsoft.

7.3.2.2.2 Software AG

To aid in the porting of DCOM to other operating systems, Microsoft has licensed it to Software AG. Software AG has already started porting it to some Unix systems, with others in the works. If DCOM becomes available on these operating systems, then it would help Microsoft's "Wintel only" image.

7.3.2.2.3 Open Group

Microsoft has submitted its part of their ActiveX and DCOM technologies to the Open Group, formerly OSF. This group will port DCOM to Java, which will improve DCOM's portability. As for how much of DCOM and ActiveX is open and how much is proprietary, only Microsoft knows that.

7.3.2.3 Java Support

Java, in its two years of public life, has become a very popular tool. Its proliferation throughout the World Wide Web (including Netscape and Microsoft's adoption of it in their browsers) is unprecedented throughout history. A lot of companies have adopted it. It is no surprise since it is supposedly a portable language, it is not just source portable, but also executable-format portable. This portability has aided its acceptance everywhere.

Java applets are already very common. Many web pages use them, and they can be very useful in an intranet environment. Java Beans are even more useful in that they can create rich and interactive web pages over the Internet or in an intranet.

7.4 Security and Encryption

Security in a distributed object system is a very important issue because:

1. There is a paradigm shift in moving from one machine to remote objects. Especially when working over the Internet, many hackers out there have the opportunity to break in to your network.
2. When there is a security breach or problem, people will not necessarily notify who is responsible, if they plan on making use of it. Thus, even if breaches are detected, they may not necessarily be repaired.

The importance of security in distributed computing over the Internet can not be deemphasized. In other words, Murphy's Law should always be applied when dealing with security in a network that services millions of users.

DCOM in Windows NT supports level C2 security from the US Department of Defense's "Orange Book". The specified CORBA Security Service allows up to B2 security for distributed objects. In order of increasing security, here are the divisions:

1. **Class D:** ***Minimal Security Protection.*** This division contains only one class. It is reserved for those systems that have been evaluated but that fail to meet the requirements for a higher evaluation class.
2. **Class C1:** ***Discretionary Security Protection.*** The Trusted Computing Base (TCB) of a class (C1) system nominally satisfies the discretionary security requirements by providing separation of users and data. It incorporates some form of credible controls capable of enforcing access limitations on an individual basis, i.e., ostensibly suitable for allowing users to be able to protect project or private information and to keep other users from accidentally reading or destroying their data. The class (C1) environment is expected to be one of cooperating users processing data at the same level(s) of sensitivity. Most Unixes are roughly about this level.
3. **Class C2:** ***Controlled Access Protection.*** Systems in this class enforce a more finely grained discretionary access control than (C1) systems, making users individually accountable for their actions through login procedures, auditing of security-relevant events, and resource isolation. Windows NT 4.0 was certified for this division.
4. **Class B1:** ***Labeled Security Protection.*** Class (B1) systems require all the features required for class (C2). In addition, an informal statement of the security policy model, data labeling, and mandatory access control over named

subjects and objects must be present. The capability must exist for accurately labeling exported information. Any flaws identified by testing must be removed.

5. **Class B2:** ***Structured Protection.*** In class (B2) systems, the TCB is based on a clearly defined and documented formal security policy model that requires the discretionary and mandatory access control enforcement found in class (B1) systems be extended to all subjects and objects in the ADP system. In addition, covert channels are addressed. The TCB must be carefully structured into protection-critical and non- protection-critical elements. The TCB interface is well-defined and the TCB design and implementation enable it to be subjected to more thorough testing and more complete review. Authentication mechanisms are strengthened, trusted facility management is provided in the form of support for system administrator and operator functions, and stringent configuration management controls are imposed. The system is relatively resistant to penetration. CORBA's security specification allows up to this protection.
6. **Class B3:** ***Security Domains.*** The class (B3) TCB must satisfy the reference monitor requirements that it mediate all accesses of subjects to objects, be tamperproof, and be small enough to be subjected to analysis and tests. To this end, the TCB is structured to exclude code not essential to security policy enforcement, with significant system engineering during TCB design and implementation directed toward minimizing its complexity. A security administrator is supported, audit mechanisms are expanded to signal security-relevant events, and system recovery procedures are required. The system is highly resistant to penetration.
7. **Class A1:** ***Verified Design.*** Systems in class (A1) are functionally equivalent to those in class (B3) in that no additional architectural features or policy requirements are added. The distinguishing feature of systems in this class is the analysis derived from formal design specification and verification techniques and the resulting high degree of assurance that the TCB is correctly implemented. This assurance is developmental in nature, starting with a formal model of the security policy and a formal top-level specification (FTLS) of the design.

Most of the above information was extracted from [DOD]. The next sections shall look at the implementations and specifications of the security features of the object technologies.

7.4.1 Distributed Object Security

Despite the existence of the CORBA Security Service, most CORBA ORBs are sorely lacking in security services. Not many have implemented the service, and even if they did, there is no guarantee that they implemented any good degree of protection (such as level C2 or above). Since DCOM comes with Windows NT 4.0 which is level C2, it automatically has many time-tested security features.

7.4.1.1 Authentication and Authorization

In Windows NT a user must log in to the machine before he can do anything, even to use DCOM objects on another machine. In Windows 95, a user does not have to log in, but then if he accesses any DCOM object, he will access it as an anonymous user. One definite advantage for system administrators using CORBA is that they do not have to worry about two orthogonal security domains; the DCOM user space is the same as that of the operating system.

In DCOM there are three sets of access permissions: Launch, Call, and Configuration access. Users with Launch access may instantiate and start new DCOM objects remotely. Users with Call access may bind to and work with objects that are already running on the remote system. Users with Configuration access may change the access permissions of that object.

No CORBA implementation integrates seamlessly with the NT operating system yet. Users must undergo a separate principal authentication scheme to be authenticated. Once this is done, the ORB retains a *Credentials* object that contains information on what the user's capabilities are.

When an invoked CORBA object needs to invoke another object, it may:

1. Use its own credentials when invoking the other method.
2. Use its client's credentials when invoking the other method.
3. Combine the client's and its own credentials in invoking the other method. The remote object will then have no way of telling whose credentials it is allowing.

The ORB may have default delegation policies, or the security-aware object can control the delegation.

7.4.1.2 Privacy and Encryption

Besides access controls, distributed object security also involves preventing sniffers from obtaining information directly off the wire. On the Internet, there is no easy way to control the route of a packet, so it is not possible to know who will see your information. It is also necessary to prevent people from modifying your data and passing it off as your messenger's. The solution to these problems all derive from data encryption.

An ORB can provide mechanisms to protect on-the-wire data. The Security service allows ORBs to work with replaceable subsystems that can encrypt on the ORB level or to integrate with a lower level encryption mechanism (such as the Secure Sockets Layer). In addition, all this happens without needing knowledge of the client or servers; it can be the ORB administrator's decision whether to use encryption or not, and for which objects.

DCOM, similar to CORBA, also allows encryption and works with replaceable subsystems. The ones who decide whether or not to use the encryption are the clients and servers, so no DCOM administrator can make a blanket "Let's encrypt everything!" statement.

Implementation-wise, DCOM is ahead of the game. Windows NT 4.0 comes with a 40 bit encryption mechanism (to comply with export laws). There are also plans to create a US/Canada version that uses 128-bit keys for stronger encryption. Whereas on the CORBA side, few implementations have implemented encryption mechanisms.

7.4.1.3 Security Holes

While DCOM has more security implemented at this time, there is a well-known bug in the DCOM architecture. This bug allows anyone anywhere to disable a DCOM machine, raising it to 100% utilization, effectively disabling any work that it was meant to do. This means that exposing a DCOM-enabled machine to the Internet is opening all sorts of potential denial-of-service attacks.

7.4.2 Component Security

7.4.2.1 ActiveX

The main problem with Microsoft's marketing of ActiveX controls is that they claim that accountability is sufficient security. This is most certainly not true. If someone were to knock on your door, tell you he is Sebastian Meretzky, shows you a birth certificate, driver's license, everything in the world to prove he is who he is, does that mean you would let him into your house unsupervised (assuming you don't know who he is)? ActiveX controls over the Internet are inherently dangerous, where you can not trust anyone.

Microsoft has released an Internet Explorer Administration Kit which allows a system administrator control where users of his domain can download ActiveX controls from. This way, browsers can be limited to downloading controls solely from trusted sources. Even so, bugs in trusted well-intentioned ActiveX controls can cause mischief in systems. It is possible to, for instance, overflow a buffer of a trusted ActiveX control. Even though this idea sounds far-fetched, this was one way Robert Morris' worm propagated itself over the Internet in 1987, and a problem like this in NCSA httpd even caused CERT to put out a security advisory on this potential problem.

7.4.2.2 Java

To exploit bugs in Java Beans or Java applets, the hacker would need to find a bug in the Java Virtual Machine for a system. There are only a few VM's out there, constantly scrutinized for security problems, the same can not be said for the thousands of ActiveX controls in existence. Even trusted Java Beans or applets can not be manipulated like this to cause problems because Java is a typesafe language.

If there are problems with the Java model, it will come primarily from trusted Java applets and Beans. The problem of accountability still must be solved. Java's sandbox model is the only safe way of solving problems *before* they happen.

Because of the newness of these technologies, there are no examples of major problems in the security of these components. It may just be a matter of time before users realize that "accountability" isn't enough. Even though JDK 1.1 support trusted applets which essentially remove security restrictions, this should be restricted to Intranet use.

Future versions of the JDK will allow better security controls; Java will at least has the capability to allow and disallow specific actions, instead of completely allowing everything like ActiveX does. For instance, for a program to print, it is not necessary to allow that program to have access to the hard drive or network connections. ActiveX, by design, can never be flexible enough for fine-grained control, but Java has the potential.

7.5 Scalability

One important thing to note is the scalability of these technologies. It is likely that DCOM and CORBA can work in a 100 object system, but what about a 1000 or 10000 object system? Unfortunately, it is very hard to tell these things apart since these technologies are too new for extensive testing. Various other aspects will be examined, though, and case studies are detailed in Section 8.

This section only applies to object technologies which use a wire protocol such as DCOM, CORBA, and Java RMI. ActiveX controls and Java applets/Beans do not need to worry about this; these technologies come into play on the local system, not with remote method invocations.

7.5.1 Object Registration and Publication

7.5.1.1 Implementation and Interface Repositories

For a DCOM client to work with a remote object, the local host needs to know the name of the server. This severely limits the maintainability of a DCOM network. As new objects get installed on the network, then every client will need to be updated. CORBA's use of implementation and interface repositories allow objects to be added the system easily and without necessarily updating the clients. If the Naming and Trader services were added to the system, then this increases the flexibility of the system; clients do not even necessarily need to have any information about the objects compiled in.

7.5.1.2 Object Activation

Most of the ORBs on the market support object activation. The servers do not need to be running; the ORB knows what executable they are located in and launch the servers. DCOM provides similar functionality; however, the client needs to know what machine the object resides on.

Many CORBA implementations also allow dynamic insertion of objects into the Interface Repository and Implementation Repository. Since DCOM has no such service, this is impossible; all the clients would have to be told when new objects arrive.

DCOM also supports object activation in Windows NT. Windows 95 however, does not support it since it would be a security risk. Therefore, servers need to be pre-run on Windows 95.

7.5.2 Fault Tolerance

CORBA also allows for fault-tolerant behavior. Since the client does not need to know where the object resides, if the ORB detects that a particular server has failed, it can redirect requests to another server. CORBA's design goal of client transparency really shines through here. As far as the client knows, all objects are persistent; they don't shut down ever. The ORB must maintain this illusion and abstraction.

With DCOM, the clients' knowledge of the location of the object precludes easy implementation of fault tolerance. There is no central ORB to tell the client where the items are.

7.5.3 Wire Protocols

7.5.3.1 Firewalls

DCOM uses a modified version of DCE RPC for communications. RPC requests are layered on top of another protocol, such as TCP or UDP. When methods are invoked remotely, they can use many different socket ports. CORBA's IIOP protocol does not work that way; all method invocations work through one port.

This allows IIOP to be used effectively through a firewall. When a new object is installed, the firewall does not have to be reconfigured to allow its packets with those new port numbers through, like with DCOM. A firewall just has to be configured to allow one port value through.

Microsoft and various CORBA vendors are working on methods to tunnel ORPC and IIOP via HTTP, respectively. While IIOP tunneling via HTTP is not very necessary, it saves the network administrators of reconfiguring the firewalls for IIOP. The problem with tunneling via HTTP is that then there can only be one-way interactions; clients can invoke from the server, but asynchronous callbacks (events and connectable objects) would not be possible.

7.5.3.2 Pinging

In DCOM, reference counts are maintained on the remote object. This means that reference counts must be correct or there will be consequences that affect other objects. The problem with this is that people often do *not* write bug-free reference counting code, and since there is also the possibility of potential network problems, servers often do not receive correct reference counting data. To solve this, DCOM uses a pinging method to determine if clients and servers are still available.

CORBA has no need for such mechanisms because CORBA objects are assumed to always be running. Reference counts are on the client side only; CORBA objects do not need to know who is using it and when. It is the ORB's job to make sure a CORBA objects are available for use, like the job of a Transaction Processing monitor. Therefore, there is also no need for pinging to determine whether or not an object or client is available; any CORBA method invocation that fails will throw an exception.

7.5.3.3 Extensibility

DCOM solely uses ORPC for method invocations. CORBA vendors may or may not use IIOP (even though they must implement it). This allows CORBA vendors to implement their own wire protocol, and in fact, CORBA vendors had to do this before CORBA 2.0. CORBA vendors would probably use IIOP or some other OMG-adopted protocol for inter-ORB communications, but for intra-ORB working, users have the option of using possibly better or more scaleable wire protocols.

7.6 Interoperability Options

One important part of these object technologies is how they can interoperate. This opens the doors to a vast array of solutions, for not only does competition create better products, users can take those products and combine them, using the best of all worlds. Some of these technologies were designed to integrate with each other (like DCOM and ActiveX), and some were not (like DCOM and CORBA). Below are possible configurations and their uses.

7.6.1 Wire Protocol Interoperability

Since CORBA 2.0, all compliant implementations must support the IIOP protocol. This ensures that all ORBs will then be able to communicate with each other. At [CORBANet], there is a demonstration of inter-ORB integration. The Cooperative Research Center for Distributed Systems Technology (DSTC), a vendor-neutral research organization, manages this showcase.

This demonstration involves using many different ORBs to interact with other ORBs in a hotel room-booking simulation. The showcase shows that it is possible to use different ORBs to interact, but it does not explain the ease or complexity involved in working with IIOP.

7.6.2 Parallel Interoperability

Parallel interoperability refers to the interoperability necessary in getting competing technologies to work together. There are various reasons for this: integration with legacy systems; integration with newly acquired information infrastructures, etc. As of now, there are too few implementations that use parallel interoperability to be able to see if it is feasible.

7.6.2.1 COM/Automation/CORBA

Various companies, including IONA Technologies and Hewlett-Packard, have added COM/CORBA integration with their ORBs. Note that this is *COM* integration, not *DCOM* integration. CORBA clients can only work with local COM objects; they can not use ORPC to talk to DCOM servers (and vice versa). Of course, there local COM objects can then use ORPC to talk to remote servers; this adds an extra level of indirection (and thus, worse performance).

For CORBA to survive on the Windows platform, interoperability with COM technologies is crucial. Not only have independent vendors created their own bridges, but the OMG is in the process on working on a specification. They have already put out a RFP on the topic.

One thing to note about CORBA was that it was designed to interoperate with other object technologies. The Dynamic Skeleton Interface on the server side is there for this purpose.

7.6.2.2 ActiveX Controls/JavaBeans

Two more competing technologies that can interoperate are ActiveX controls and JavaBeans. JavaBeans was designed to interoperate with ActiveX controls, so the design was planned out carefully in the expectations of a bridge. They both support many of the same features, including downloadability and scriptability. However, no one has released a bridge for use by the public (although Sun is currently working on one).

7.6.3 Orthogonal Interoperability

In addition to parallel interoperability is orthogonal interoperability. For many of these solutions, the technologies were designed to enable interoperability. At first glance at these solutions, one can easily see how companies are beginning to realize the potential of the WWW, and enable better object solutions with it. The main idea of these integration solutions is visualized in Figure 31.

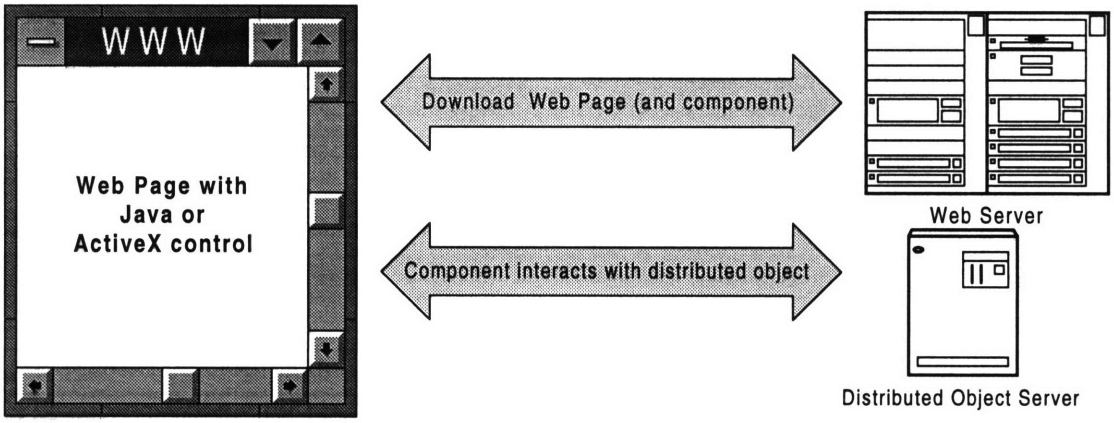

WWW
Web Page with Java or ActiveX control
Download Web Page (and component)
Component interacts with distributed object
Web Server
Distributed Object Server

Figure 31 Orthogonal Interoperability

The main idea of this paradigm is that users only need to install *one* client: the web browser. This web browser can download components off a web page and those components can use DCOM, CORBA, or RMI to communicate with remote objects. This allows easier distribution of services and maintainability of clients.

7.6.3.1 Java and CORBA

The combination of Java and CORBA is likely to be used widely in the future. One reason for this is that they can run on most platforms, and Netscape is bundling a CORBA runtime in their newest browser. CORBA implementations in Java can run on any Java VM, unlike DCOM. This is a great advantage in distribution and again, in working with legacy systems.

7.6.3.2 ActiveX Controls and DCOM

Microsoft is promoting the use of ActiveX and DCOM together for distributed computing. ActiveX and DCOM share the same roots in OLE, so they can integrate well together without kludging interoperability. On a Windows-only intranet, this would be the logical choice to use. This may not be as good a solution on a large heterogeneous network (like the Internet).

7.6.3.3 JavaBeans and RMI

Using a Java Bean to communicate using RMI is as natural as using ActiveX controls communicating with DCOM. This combination will work best in a small homogenous environment of machines that all run Java VMs. Since both client and server run Java, remote Java objects can be passed around without worrying about conforming to IDL mapping, which can be useful in pass-by-value situations.

7.6.3.4 Java and DCOM

Java can work with DCOM, but the integration is not very smooth, especially with distributed objects. Not only that, Java integration with DCOM is a Microsoft modification of their virtual machine; it is not in the specification. There is one very big problem with working with downloading Java applets: Java currently can not specify a location for DCOM servers like C++ can. This means that unless a client machine has remote object location information *a priori* to viewing the web page, it will not be able to choose which server to work with.

In effect, Java applet integration with DCOM is a lost cause with current technology and tools.

8. Existing Projects

While talking about object technologies and their implementations is good at a fundamental level, it is essential to examine concrete examples of installations and their successes. As of now, the majority of installed distributed object architectures use CORBA. DCOM and Java RMI are both too new.

8.1.1 United Kingdom Immigration Service

The United Kingdom Immigration service runs the largest CORBA implementation in the world. It was implemented by ICL Enterprises, using their own DAIS ORB. The problem that this implementation solves involves large scale distribution of images and documents.

Every year, 10 million people seek entry to the United Kingdom. Some people are not allowed to enter the country, but trying to detect these people was tedious and not always reliable. With the installation the Immigration Service Suspect Index Computer System, this task has improved.

The Suspect Index (SI) is a book that contains 10,000 entries; each entry contains information about a particular person that should not enter the country. This book was limited to 300 pages and only contained the most important information on people. This limited its usefulness in screening passengers. Maintaining the information in the book (and then redistributing it) was a labor-intensive process that drew officer away from their main tasks.

The CORBA solution replaces the use of this book with a distributed document and image management system. They use two UNIX servers running an ORACLE 7 relational database management system to hold all the data. The system uses DAIS to handle all movement of images and data cross the network. The DAIS portion facilitates:

- Scalability
- Heterogeneous platforms
- Run-time flexibility
- Distributed applications
- Legacy integration
- Portable multimedia data
- Application resilience and high availability
- Remote management of customer applications

When a visitor arrives at the immigration desk, the officer inputs the person's name to the computer, and it checks to see if he is in the Suspect Index. If more information is required, the officer can then look up more information using a personal computer in the back office.

The system has run for 12 months with no total system outages. It has processed 10 million passports, and over 15,000 images were downloaded at an average of 12 seconds per image.

8.1.2 Cariplo Bank

Cariplo S.p.A., a bank in Italy, needed to improve its home banking system. The old system gave home bankers the same user interface that the tellers at the bank used, thus few home users took advantage of it. The solution they implemented was in 1996 and involved:

1. ActiveX Controls on the client side
2. DCOM on the server side (since DCOM did not exist on Windows 95)
3. HTTP as the communication protocol between client and server.

With the new system, a user would Microsoft Internet Explorer 3.0 as the web browser. It would download and work with ActiveX controls that provide rich user interfaces (tabbed dialog boxes, etc.) that would allow them easy access to their accounts. The ActiveX control would communicate with the accounts server using http. The ActiveX controls used the Secure Sockets Layer to authenticate and encrypt the ISAPI commands. Now that there is a Windows 95 implementation of DCOM, they are planning to use DCOM instead.

The web server would use ISAPI to communicate with various COM business objects that provide different functions. They used free-threading to achieve maximum performance and scalability. The fact that they were COM objects makes the system more modular, and allows future flexibility of the deployment of the business objects.

The home banking application needed to use the existing mainframe system, but it was one design constraint was to leave the mainframe system unmodified. This meant that extra metadata, like security credentials, access logs, and home banking specific fees, had to be stored elsewhere. The implementers chose Microsoft SQL Server as their database, and the integration between the database and mainframe system was handled by the business objects.

Both ODBC and the mainframe's LUA2 protocols are connection oriented, meaning they were expensive to work with. Thus, they implemented a Pool Manager component that manages a number of connections. As other components requires connections, they would request it from the Pool Manager. These Pool Managers had nothing to do with the business objects, so to constrain them to the same machine as the business objects was not a smart solution. Instead, they used DCOM for communication between the components, to allow scalability and ease of implementation.

In the future, if DCOM can be used to integrate the client-side components. The client's ActiveX controls can then communicate directly with business objects or pool managers instead of having to work through the web server. This also gives the whole system a better position if there arises a need to make extensive changes in the future.

8.1.3 Wells Fargo

In the 1980s, many banks began offering brokerage and mutual funds accounts to its customers. Wells Fargo was one of them. However, at Wells Fargo, the account data was stored on different systems, so when a customer wanted information on all his accounts, there was no integrated way of getting to it. These systems used VT100 or 3270 emulators to access mainframes with the data, and the user interfaces were not very uniform. In fact, there was no way to find out list what accounts the customer had with the bank without querying each database.

To solve this problem, Wells Fargo decided to use Digital's ObjectBroker ORB to integrate all these subsystems. A business object was developed to access and integrate the data from the various mainframes, and a user interface was designed to access the business object. In the end, they created a system that allowed representatives to uniformly work with all of a customer's accounts. The system was placed in full production in 104 days, in January 1994.

In March 1995, Wells Fargo decided to provide online banking services over the WWW. Sixty days later, they finished the project. The project took so short because they were able to directly leverage the business objects developed in the previous project. If it weren't for the security issues, it would have taken shorter. All they would have had to do was make a web server an ObjectBroker client. Most of the development efforts were put into maintaining security.

The web-based banking service has now grown to over 100,000 enrolled customers.Its ORB servers were processing 200,000 business object invocations per day, and at peak periods there were as many as 300 simultaneous Internet customer. This project exemplifies the production-scale capabilities of CORBA. Future services to customer can be added with less effort because the business objects can be reused.

8.1.4 MITRE

The Air Force sponsored research on DOMIS, the Distributed Object Management Integration System. The research was conducted by the MITRE corporation. The project involved integration of legacy subsystems into a distributed object management environment. They used CORBA to do this.

The Air Force's Contingency Theater Air Control System (TACS) Automated Planning System (CTAPS) consisted of about 2.5 million lines of code. The DOMIS project has already integrated three subsystems, containing roughly 500,000 lines of C, C++, Ada, SQL, and Pro C. Each subsystem was encapsulated as a single CORBA object, invoked through a CORBA interface. At first, they used the HyperDesk Distributed Object Management System, but eventually ported their work to IONA Technologies' Orbix ORB. There is a detailed description of their work at [DOMISImpl].

The research is still going on; the Air Force is converting more CTAPS subsystems over.

9. Example CORBA Implementation

This section talks about an implementation of a simple Mandelbrot set calculation program using Visigenic's ORB. Calculating the Mandelbrot set takes a non-trivial amount of time; this implementation distributes calculations to other computers.

This program was implemented in Visual C++ 4.2b on a Pentium/133 running Windows 95.

9.1 Mandelbrot Set and Fractals

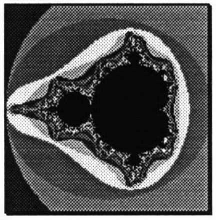


Figure 32 Mandelbrot Set

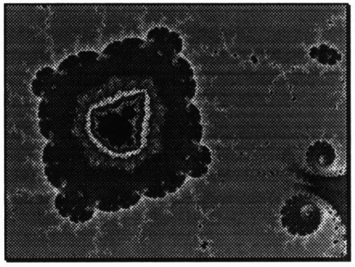


Figure 33 Mandelbrot Set zoom in

Fractal mathematics involves the generation of sets given from a recursive formula. The characteristic equation of the Mandelbrot set is:

$$z_{n+1} = z_n^2 + c$$

z and c are complex numbers. Given a point c, to determine if it is in the Mandelbrot set, the computer would start by assigning a starting value to z_0 (usually zero). The computer then iterates z a prespecified number of times or until its magnitude goes higher than 2 (which indicates it will approach infinity).

If z does approach infinity, the point c is not in the set. The point is then assigned a color, usually the color is the n mod the maximum number of colors to use. If the point does not approach infinity, it is assigned a neutral color, usually black.

The Mandelbrot set is easily recognizable; most computer users have probably seen it. Figure 32 is an example of a Mandelbrot set. Mandelbrot sets are infinitely zoomable, and each zoom in shows more and more detail. Figure 33 shows another view of the Mandelbrot set.

9.2 Example Design

The goal of this program was not to create an easily extensible program for future reuse and enhancement. Rather, this was supposed to be a "bare bones" type of program to illustrate how CORBA can be used in distributed object programming. Thus, the design was not thought out to accommodate extensive modifications.

9.2.1 Overview of Example Design

There are basically two types of objects in this program, the *Display* object and the *Fractal* object. The *Display* object assumes that the *Fractal* server are already started. Whether or not they are on separate machines or not does not matter. When the local Display server starts, it will wait for the user to indicate what coordinates and at what resolution they want the image. It must also know how many servers there are available (not very robust, but to the point).

When the Display server starts, it will register the Display object so Visigenic's Smart Agent will recognize its existence. It will then send requests to the Fractal Servers to calculate different portions of the Mandelbrot image. It divides the image into horizontal stripes, and asks each fractal server to calculate it. When it asks the Fractal servers to calculate the image portions, it also passes each Fractal object a reference to itself.

As each Fractal object finishes calculating line, it invokes methods on the Display object, passing it image data. The Display object will then display image data as it receives it. Figure 34 illustrates this design.

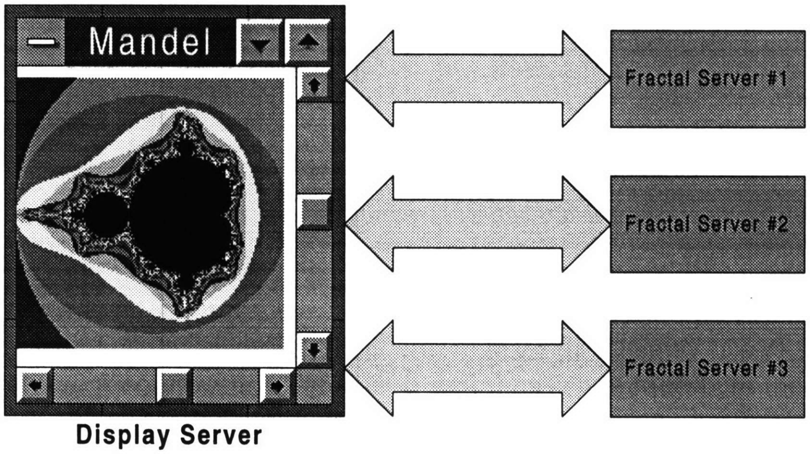


Figure 34 Overall design of the Mandelbrot viewer

9.2.2 Example Interfaces

```
typedef long Pixel;            // Windows RGBQUAD format
typedef sequence<Pixel> SeqPixel;
interface Display{
   oneway void PutPixels(in long x, in long y,
                         in long nWidth, in long nHeight,
                         in SeqPixel pixels);
   oneway void PutPixel(in long x, in long y, in Pixel pixel);
   oneway void Repaint();
};
```

Figure 35 *display.idl*

Figure 35 and Figure 36 are the listings for the IDL files used. In *display.idl*, the *Pixel* and *SeqPixel* data types are defined. The *Pixel* data type is simply a *long*, which in CORBA IDL is a 4 byte integer. This fits in with Win32's RGBQUAD format used for displaying bitmaps. *SeqPixel* is simply a sequence of pixels.

The interface *Display* has three methods. *PutPixels* takes in a rectangle (the upper left corner plus the width and height) and a sequence of pixels. The sequence of pixels contains the image data to draw into the a memory buffer of the *Display* object. *PutPixel* is a shorter form of *PutPixels*, it takes one pixel instead of a block. Finally, *Repaint* tells

```
#include "display.idl"
interface Fractal {
   oneway void calculate(in double fpointX, in double fpointY,
                         in double fpointWidth,
                         in double fpointHeight,
                         in long nXcorner,
                         in long nYcorner,
                         in long nWidth,
                         in long nHeight,
                         in Display myDisplay);
};
```

Figure 36 fractal.idl

the *Display* object to transfer the contents of the memory buffer to the display of the machine that is running the object.

The *Fractal* interface is even more straightforward than the *Display* interface. The *Fractal* interface only has one method, *calculate*. The first four arguments represent the a rectangle. This rectangle represents the set of numbers that the *Fractal* object must determine are in the Mandelbrot set or not. The next four coordinates represent a screen region which the *Fractal* object must write to. They also tell the *Fractal* object what resolution it needs to calculate. The last argument is a reference to a *Display* object which the *Fractal* object will work with. Note that the *Display* object is an *in* parameter, not an *in/out* parameter since the *calculate* method does not modify the variable; it merely calls the *Display* object's methods.

All of these methods are *oneway* methods for speed purposes. *oneway* methods do not check for return values or even if the method invocation succeeded. The client can simply call the method and continue on without waiting for a response.

9.2.3 Example Implementation Details

There are two separate executables running in this distributed object network, the *Display* and *Fractal* servers.

9.2.3.1 Display Server Implementation

The *Display* server uses the Microsoft Foundation Classes 4.2. The relevant files in the project are:

display_c.cpp	Generated by IDL compiler
display_s.cpp	Generated by IDL compiler
fractal_c.cpp	Generated by IDL compiler
display_impl.cpp	**Implementation of the *Display* interface**
DispSrv.cpp	Generated by MFC AppWizard, modified
DispSrvDoc.cpp	Generated by MFC AppWizard, modified
DispSrvView.cpp	Generated by MFC AppWizard, modified
MainFrm.cpp	Generated by MFC AppWizard, modified
orb_r.lib	Required to work with Visigenic ORB DLL

The most important file to look at is the *display_impl.cpp* file. This is the implementation of the *Display* interface. The methods do not need to be explained, since none of them are CORBA specific. They are either straight C++ or Windows graphics specific.

The main CORBA specific areas involve initializing the *Display* implementation and making the function call to the remote *Fractal* object. In *CDispSrcDoc::DoImage()*, the program initializes the ORB and constructs the *Display* implementation. It calls the basic object adapter's *obj_is_ready()* method, to let the ORB know that our object is initialized.

The *Display* server assumes (for simplicity's sake) that the number of *Fractal* implementations on the network is passed in the command line. This makes it easy to bind all the available *Fractal* objects. Once all the *Fractal* objects have been bound, the program then calls the *calculate* method of each of the *Fractal* objects with the appropriate parameters.

The program then goes into the standard Windows message loop, doing nothing else unless its *Display* implementation receives method invocations from the *Fractal* implementations on the net.

9.2.3.2 Fractal Server Implementation

The *Fractal* server uses the Microsoft Foundation Classes 4.2. The relevant files in the project are:

display_c.cpp	Generated by IDL compiler
fractal_c.cpp	Generated by IDL compiler
fractal_s.cpp	Generated by IDL compiler
mandel_impl.cpp	**Implementation of the *Fractal* interface**
main.cpp	The main program
orb_r.lib	Required to work with Visigenic ORB DLL

The *main()* function is simple. Since this is essentially just a server without a need for a user interface or anything, the *main()* function just initializes the object with the name that's passed on the command line. It then calls the basic object adapter's *obj_is_ready()* to let the ORB know the object is ready, and then *impl_is_ready()* to let it know that the ORB know that all the objects have been initialized.

Mandel_Impl::calculate() is also straightforward. There is very little there that is CORBA specific. The function does not even need to know that the *Display* variable is an interface to a CORBA object. The function then iterates and calculates the color of the points in the region it was told to calculate. After each point is determined, it calls the *Display* object's *PutPixel* routine. After every two lines, it calls the *Repaint()* method so the window on the user's screen will update. This is a crude way of doing it, but it is purposely simple to illustrate the ease of use of CORBA.

Note that the magic of CORBA is fully shown in *Fractal::calculate*'s last parameter, which is an object reference to a *Display* object. This is where the automatic marshaling of parameters is truly important.

9.2.4 *Display* Implementation Code

9.2.4.1 display.idl

```
typedef long Pixel;       // Windows RGBQUAD format
typedef sequence<Pixel> SeqPixel;

interface Display{
      oneway void PutPixels(in long x, in long y,
                                         in long nWidth, in long nHeight,
                                         in SeqPixel pixels);
      oneway void PutPixel(in long x, in long y, in Pixel pixel);
      oneway void Repaint();
};
```

9.2.4.2 display_impl.cpp

```
// Display

#include <corba.h>
#include <afxwin.h>
#include <afxmt.h> // multithreading
#include <windows.h>
#include "display_s.hh"
#include "display_impl.h"
#include "dispsrvdoc.h"

void Display_Impl::PutPixels(CORBA::Long x, CORBA::Long y,
                         CORBA::Long nWidth, CORBA::Long nHeight,
                         const SeqPixel & pixels){
   // Make it reentrant
   CCriticalSection myCriticalSection;
   myCriticalSection.Lock();

   for (CORBA::Long ypoint = 0; ypoint < nHeight; ++ypoint){
      for (CORBA::Long xpoint = 0; xpoint < nWidth; ++xpoint){
         int iIndex = GetIndex(xpoint + x, ypoint + y);

         Pixel mycolor = pixels[xpoint + ( ypoint)* nWidth];
         m_pBits[iIndex] = mycolor;
      }
   }

   myCriticalSection.Unlock();
};

long Display_Impl::GetIndex(int x, int y){
   BITMAPINFOHEADER *pBI = (BITMAPINFOHEADER *) m_pBMI;

   long yoffset = pBI->biWidth * y;
   long xoffset = x;

   return xoffset + yoffset;

}

void Display_Impl::Repaint(){
   AfxGetMainWnd()->Invalidate(FALSE);
}

void Display_Impl::PutPixel(CORBA::Long x, CORBA::Long y, Pixel pixel){
   // Make it reentrant
   CCriticalSection myCriticalSection;
   myCriticalSection.Lock();

   m_pBits[GetIndex(x, y)] = pixel;
   myCriticalSection.Unlock();
}

Display_Impl::Display_Impl(int nWidth, int nHeight, char *object_name)
                         : _sk_Display(object_name){
   // nWidth and nHeight are the width and height of the display area

   m_pBMI = new BITMAPINFO;

   BITMAPINFOHEADER *pBI = (BITMAPINFOHEADER *) m_pBMI;

   pBI->biSize = sizeof(BITMAPINFOHEADER);
   pBI->biWidth = nWidth;
   pBI->biHeight = -nHeight;
   pBI->biPlanes = 1;
   pBI->biBitCount = 32;
   pBI->biCompression = BI_RGB;
   pBI->biSizeImage = 0;
   pBI->biXPelsPerMeter = 0;
   pBI->biYPelsPerMeter = 0;
   pBI->biClrUsed = 0;
   pBI->biClrImportant = 0;

   m_pBits = new Pixel[nWidth * (nHeight + 1)];
}

Display_Impl::~Display_Impl(){
   delete m_pBMI;
   delete [] m_pBits;
}
```

9.2.4.3 DispSrvDoc.h

```
// DispSrvDoc.h : interface of the CDispSrvDoc class
//
/////////////////////////////////////////////////////////////////////////////

class CDispSrvDoc : public CDocument
{
protected: // create from serialization only
   CDispSrvDoc();
   DECLARE_DYNCREATE(CDispSrvDoc)

// Attributes
public:
   Display_Impl *m_pDisplayImpl;

// Operations
public:

   Pixel *GetBitsAddress() const;
   int GetHeight() const;
   int GetWidth() const;
   BITMAPINFO *GetBitmapInfo() const;
   void NewParam();

// Overrides
   // ClassWizard generated virtual function overrides
   //{{AFX_VIRTUAL(CDispSrvDoc)
   public:
   virtual BOOL OnNewDocument();
   virtual void Serialize(CArchive& ar);
   //}}AFX_VIRTUAL

// Implementation
public:
   virtual ~CDispSrvDoc();
#ifdef _DEBUG
```

```
   virtual void AssertValid() const;
   virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
   //{{AFX_MSG(CDispSrvDoc)
      // NOTE - the ClassWizard will add and remove member functions here.
      //    DO NOT EDIT what you see in these blocks of generated code !
   //}}AFX_MSG
   DECLARE_MESSAGE_MAP()

private:
   double m_fStartX, m_fStartY, m_fpointWidth, m_fpointHeight;
   int m_nWidth, m_nHeight;

   void DoImage(CORBA::Double fpointX,
         CORBA::Double fpointY,
         CORBA::Double fpointWidth,
         CORBA::Double fpointHeight,
         CORBA::Long nWidth,
         CORBA::Long nHeight);
};

/////////////////////////////////////////////////////////////////////////////
```

9.2.4.4 DispSrvDoc.cpp

```
// DispSrvDoc.cpp : implementation of the CDispSrvDoc class
//

#include "stdafx.h"
#include "DispSrv.h"
#include "..\display_s.hh"
#include "display_impl.h"
#include "..\fractal_c.hh"
#include "DispSrvDoc.h"
#include "SelectParamDlg.h"
#include "corba.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CDispSrvDoc

IMPLEMENT_DYNCREATE(CDispSrvDoc, CDocument)

BEGIN_MESSAGE_MAP(CDispSrvDoc, CDocument)
   //{{AFX_MSG_MAP(CDispSrvDoc)
      // NOTE - the ClassWizard will add and remove mapping macros here.
      //    DO NOT EDIT what you see in these blocks of generated code!
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CDispSrvDoc construction/destruction

CDispSrvDoc::CDispSrvDoc()
{
   m_fStartX = -2;
   m_fStartY = -1.5;
   m_fpointWidth = 3;
   m_fpointHeight = 3;
   m_nWidth = 150;
   m_nHeight = 150;
   m_pDisplayImpl = NULL;
}

CDispSrvDoc::~CDispSrvDoc()
{
   if (m_pDisplayImpl)
      delete m_pDisplayImpl;
}

BOOL CDispSrvDoc::OnNewDocument()
{
   // Initialize CORBA object
   CORBA::ORB_var orb = CORBA::ORB_init(__argc, __argv);
   CORBA::BOA_var boa = orb->BOA_init(__argc, __argv);

   if (m_pDisplayImpl){
      boa->deactivate_obj((Display *) m_pDisplayImpl);
      delete m_pDisplayImpl;
      m_pDisplayImpl = NULL;
   }

   m_pDisplayImpl = new Display_Impl(100, 100, "MyName");

   return TRUE;
}

void CDispSrvDoc::NewParam()
{
   CSelectParamDlg dlg;

   dlg.m_fStartX = m_fStartX;
   dlg.m_fStartY = m_fStartY;
   dlg.m_fEndX = m_fStartX + m_fpointWidth;
   dlg.m_fEndY = m_fStartY + m_fpointHeight;
   dlg.m_nWidth = m_nWidth;
   dlg.m_nHeight = m_nHeight;

   if (!dlg.DoModal())
      return ;

   m_fStartX = dlg.m_fStartX;
   m_fStartY = dlg.m_fStartY;
   m_fpointWidth = dlg.m_fEndX - dlg.m_fStartX;
   m_fpointHeight = dlg.m_fEndY - dlg.m_fStartY;
   m_nWidth = dlg.m_nWidth;
   m_nHeight = dlg.m_nHeight;

   DoImage(dlg.m_fStartX, dlg.m_fStartY,
         dlg.m_fEndX - dlg.m_fStartX, dlg.m_fEndY - dlg.m_fStartY,
         dlg.m_nWidth, dlg.m_nHeight);
}

void CDispSrvDoc::DoImage(CORBA::Double fpointX,
         CORBA::Double fpointY,
         CORBA::Double fpointWidth,
         CORBA::Double fpointHeight,
         CORBA::Long nWidth,
         CORBA::Long nHeight){

   Fractal *Fractals[10];
   // limit of 10

   int nObjects = 1;
   if (__argc == 2)
```

```
      nObjects = __argv[1][0] - 48;
   // Only take first digit

   // Initialize CORBA object
   CORBA::ORB_var orb = CORBA::ORB_init(__argc, __argv);
   CORBA::BOA_var boa = orb->BOA_init(__argc, __argv);

   if (m_pDisplayImpl){
      boa->deactivate_obj((Display *) m_pDisplayImpl);
      delete m_pDisplayImpl;
   }

   m_pDisplayImpl = new Display_Impl(nWidth, nHeight, "MyName");

   boa->obj_is_ready(m_pDisplayImpl);

   // The number of Fractal servers on the network out there is in the command
line.
   // We assume they are number "1", "2", etc. Max 10
   for (int i = 1; i <= nObjects; i++){
      try{
         CString cstrName = char(i + 48);
         Fractals[i] = Fractal::_bind((const char *)cstrName);
      }
      catch(CORBA::Exception& excep) {
         AfxMessageBox("Error binding to server");
         return;
      }
   }

   for (i = 1; i <= nObjects; i++){
      try {
         Fractals[i]->calculate(fpointX, fpointY + (i - 1) * fpointHeight /
nObjects,
                              fpointWidth, fpointHeight / nObjects,
                              0, (i - 1) * nHeight / nObjects,
                              nWidth, nHeight / nObjects,
                              (Display *)m_pDisplayImpl);
         AfxGetMainWnd()->Invalidate(FALSE);
      }
      catch(CORBA::SystemException& excep) {
         AfxMessageBox("Error invoking method");
      }
   }
}

/////////////////////////////////////////////////////////////////////////////
// CDispSrvDoc commands

int CDispSrvDoc::GetWidth() const
{
   BITMAPINFOHEADER *pBI = (BITMAPINFOHEADER *)m_pDisplayImpl->m_pBMI;

   return pBI->biWidth;
}

int CDispSrvDoc::GetHeight() const
{
   BITMAPINFOHEADER *pBI = (BITMAPINFOHEADER *)m_pDisplayImpl->m_pBMI;

   return abs(pBI->biHeight);
}

Pixel *CDispSrvDoc::GetBitsAddress() const
{
   return m_pDisplayImpl->m_pBits;
}
```

```
BITMAPINFO *CDispSrvDoc::GetBitmapInfo() const
{
   return m_pDisplayImpl->m_pBMI;
}
```

9.2.5 *Fractal* Implementation Code

One extra point about this implementation needs to be mentioned. Visigenic's ORB at the time had trouble working with Visual C++ 4.2's implementation of the Standard C++ Class Library (they have said this problem will be fixed in the next version). Therefore, this implementation could not use the built-in *complex* data type. Instead, a partial implementation was used, by doing some discretionary cutting and pasting from the Visual C++'s implementation.

9.2.5.1 fractal.idl

```
#include "display.idl"

interface Fractal {
      oneway void calculate(in double fpointX, in double fpointY,
                            in double fpointWidth,
                            in double fpointHeight,
                            in long nXcorner,
                            in long nYcorner,
                            in long nWidth,
                            in long nHeight,
                            in Display myDisplay);
};
```

9.2.5.2 main.cpp

```
#include <windows.h>
#include "..\fractal_s.hh"
#include "..\display_c.hh"
#include "mandel_impl.h"

int main(int argc, char *const *argv)
{
   CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
   CORBA::BOA_var boa = orb->BOA_init(argc, argv);

   char *name;

   if (argc != 2)
      name = "1";
   else
      name = argv[1];

   Mandel_Impl mandel(name);

   boa->obj_is_ready(&mandel);

   boa->impl_is_ready();

   return(1);
}
```

9.2.5.3 mandel_impl.cpp

```
// mandel_impl.cpp

#include "..\fractal_s.hh"
#include "..\display_c.hh"
#include "mandel_impl.h"
```

```
#include "complexpart.h"  // Incompatible with Standard C++ library!

Mandel_Impl::Mandel_Impl(char *object = NULL) : _sk_Fractal(object){
   cout << "Object Initialized" << endl;
}

void Mandel_Impl::calculate(CORBA::Double fpointX,
          CORBA::Double fpointY,
          CORBA::Double fpointWidth,
          CORBA::Double fpointHeight,
          CORBA::Long nXcorner,
          CORBA::Long nYcorner,
          CORBA::Long nWidth,
          CORBA::Long nHeight,
          Display_ptr myDisplay){
   // This will calculate the mandelbrot set at

   if (!ReadMapFile("default.map"))
   {
      cout << "Could not read color map file" << endl;
      return ;
   }

   cout << "Calculate called with " << fpointX << ", " << fpointY << "; " <<
                              fpointWidth << ", " << fpointHeight << "; "
                              << nWidth << ", " << nHeight << endl;

   SeqPixel FrameBuffer;
   CORBA::Double ftempX = fpointX, ftempY = fpointY,
                 fXinterval = fpointWidth / nWidth,
                 fYinterval = fpointHeight / nHeight;
   long lIterations;

   FrameBuffer.length(nWidth * nHeight);

   for (CORBA::Long y = 0; y < nHeight; ++y, ftempY += fYinterval){
      ftempX = fpointX;
      for (CORBA::Long x = 0; x < nWidth; ++x, ftempX += fXinterval){
         lIterations = MandelCalc(ftempX, ftempY);
         FrameBuffer[x + y * nWidth] = m_ColorSpace[lIterations % 256];
         myDisplay->PutPixel(x + nXcorner, y + nYcorner, FrameBuffer[x + y *
nWidth]);
      }
      if (y % 2)
         myDisplay->Repaint();
      cout << "Plotted Line " << y << endl;
   }

   myDisplay->PutPixels(nXcorner, nYcorner, nWidth, nHeight, FrameBuffer);
   myDisplay->Repaint();
};

long Mandel_Impl::MandelCalc(CORBA::Double x, CORBA::Double y){
   int i;

   complexPart<CORBA::Double> c(x,y);
   complexPart<CORBA::Double> z(0, 0);

   // Loop and see how many iterations it takes z to go out of bounds, under
   // m_lIterations
   for (i = 1; i < 1000; i++){
      z = z * z + c;
      if (norm(z) >= 4)
         return i;
   }

   return 0;
}

int Mandel_Impl::ReadMapFile(char *name){
   FILE *file;
```

```
    char temp[300];
    file = fopen(name, "ra");

    if (file){
        for (int i = 0; i < 256; ++i){
            char red, green, blue;

            fscanf(file, "%d%d%d\n", &red, &green, &blue);

            m_ColorSpace[i] = long((long(long(blue) << 16) +
                                   long(long(green) << 8) +
                                   long(red)));
        }
    }
    else
    {
        return 0;
    }
    fclose(file);
    return 1;
}
```

9.2.5.4 complexPart.h

```
// complexPart.h
// Partial implementation of a complex data type that works with Visigenic's
ORB
template <class _type>
class complexPart{

public:

/*  complexPart<_type> operator*(const complexPart<_type> &op){
        return op;
    }
*/
    complexPart<_type> operator+(const complexPart<_type> &op){
        return complexPart<_type>(real + op.real, imag + op.imag);
    }

    _type getreal(){
        return real;
    }

    _type getimag(){
        return imag;
    }

    complexPart<_type>(_type r, _type i){
        real = r;
        imag = i;
    }

friend complexPart<_type> operator*(complexPart <_type> &left,
complexPart<_type> &right);

private:
    _type real;
    _type imag;

};

// Helper function
template<class _type>
inline double norm(complexPart<_type> &val)
{
    return (val.getreal() * val.getreal() + val.getimag() * val.getimag());
}

template<class _type>
inline complexPart<_type> operator*(complexPart <_type> &left,
complexPart<_type> &right){
   return complexPart<_type>(left.real * right.real - left.imag * right.imag,
                             left.imag * right.real + left.real * right.imag);
}
```

9.3 CORBA Implementation Conclusion

The implementation of this example was rather straightforward. Working off the documentation and examples provided by Visigenic as well as other references on CORBA, it was simple to compile and run this example. If the ORB was more integrated with Visual C++'s Integrated Development Environment (as such, it may be with Borland's JBuilder), then this project would have been a lot easier to implement without impedance mismatches.

One reason why there were no problems is probably because of the simplicity of CORBA. It was designed to allow low-level objects to communicate with each other. Unlike DCOM, as we'll see in the next section, CORBA did not have any extra baggage when it was developed.

10. Example COM Implementation

The COM implementation is very similar to the CORBA implementation, so there is no need to talk about the overall view and design. Like the CORBA implementation, it uses two objects, a display object and a fractal object. The final implementation was written using the ActiveX Template Library for the fractal object, and just straight COM for the client.

10.1 Example Design

Notice that there are interfaces: *IMandel* and *IDisplay*. This parallels the CORBA design. However, instead of passing in an interface pointer into the *IMandel::calculate* method, the actual object uses the connection point technology of COM. This is not indicated in *connect.idl* below, because MIDL files only describe interfaces, not objects. In the actual class header file, the interfaces supported by an object are then described.

10.1.1 Interfaces

10.1.1.1 connect.idl

```
// Connect.idl : IDL source for Connect.dll
//

// This file will be processed by the MIDL tool to
// produce the type library (Connect.tlb) and marshalling code.

        [
                object,
                uuid(CCE84211-DB31-11CF-9D75-00A0C90391D3),
                dual,
                helpstring("IMandel Interface"),
                pointer_default(unique)
        ]
        interface IMandel : IDispatch
        {
                import "oaidl.idl";
                HRESULT calculate([in] double fpointX,
                                       [in] double fpointY,
                                       [in] double fpointWidth,
                                       [in] double fpointHeight,
                                       [in] long nXcorner,
                                       [in] long nYcorner,
                                       [in] long nWidth,
                                       [in] long nHeight);
        };

        [
                object,
                uuid(CCE84212-DB31-11CF-9D75-00A0C90391D3),
                dual,
                helpstring("IDisplay Interface"),
                pointer_default(unique)
        ]
        interface IDisplay : IUnknown
        {
                import "oaidl.idl";
                HRESULT PutPixel(long x, long y, long color);
                HRESULT Repaint();
        };
```

```
[
	uuid(CCE8420F-DB31-11CF-9D75-00A0C90391D3),
	version(1.0),
	helpstring("Connect 1.0 Type Library")
]
library CONNECTLib
{
	importlib("stdole32.tlb");

	[
		uuid(CCE84215-DB31-11CF-9D75-00A0C90391D3),
		helpstring("Mandel Class")
	]
	coclass CoMandel
	{
		[default] interface IMandel;
		[default, source] interface IDisplay;
	};
};
```

10.1.2 Objects

The *CMandel* class uses the ActiveX Template Library. Much of the work is abstracted away by using templates, macros, and inheriting from base classes. A thorough discussion of each of the classes is beyond the scope of this thesis. Note that it inherits connection point functionality as well as basic COM object functionality.

The *CDisplay* class is more of a bare bones COM object. It merely inherits the interface from *IDisplay*, and implements all the required interfaces.

10.1.2.1 Mandel.h

```
#include "connres.h"		// main symbols

/////////////////////////////////////////////////////////////////////////////
// CMandel

const int nMaxSessions = 10;

class CMandel :
	public IDispatchImpl<IMandel, &IID_IMandel, &LIBID_CONNECTLib>,
	public IConnectionPointContainerImpl<CMandel>,
	public IConnectionPointImpl<CMandel, &IID_IDisplay,
CComDynamicUnkArray>,
	public ISupportErrorInfo,
	public CComObjectRoot,
	public CComCoClass<CMandel,&CLSID_CoMandel>
{
public:
	CMandel()
	{
	}

BEGIN_COM_MAP(CMandel)
	COM_INTERFACE_ENTRY2(IDispatch, IMandel)
	COM_INTERFACE_ENTRY(IMandel)
	COM_INTERFACE_ENTRY(ISupportErrorInfo)
	COM_INTERFACE_ENTRY_IMPL(IConnectionPointContainer)
END_COM_MAP()

	DECLARE_REGISTRY_RESOURCEID(IDR_Random)

// Connection Point
	BEGIN_CONNECTION_POINT_MAP(CMandel)
```

```
             CONNECTION_POINT_ENTRY(IID_IDisplay)
      END_CONNECTION_POINT_MAP()

// ISupportsErrorInfo
      STDMETHOD(InterfaceSupportsErrorInfo)(REFIID riid);

// IMandel
      STDMETHOD(calculate)(double fpointX,
                                       double fpointY,
                                       double fpointWidth,
                                       double fpointHeight,
                                       long nXcorner,
                                       long nYcorner,
                                       long nWidth,
                                       long nHeight);

private:
      long MandelCalc(double x, double y);
      int ReadMapFile(char *name);
      long m_ColorSpace[256];
};
```

10.1.2.2 CDisplay.h

```
#include "stdafx.h"
#include <windows.h>
#include <ole2.h>
#include <ocidl.h>
#include <olectl.h>
#include <objbase.h>
#include <atlbase.h>
extern CComModule _Module;
#include <atlcom.h>
#include "..\connect.h"

/////////////////////////////////////////////////////////////////////////////
// CDisplay

#ifndef CDISPLAY

#define CDISPLAY

class CDisplay :
      public IDisplay
{
public:
      CDisplay() {
             m_nRef = 0;
             m_pBMI = new BITMAPINFO;
             BITMAPINFOHEADER *pBI = (BITMAPINFOHEADER *)m_pBMI;

             pBI->biSize = sizeof(BITMAPINFOHEADER);
             pBI->biWidth = 100;
             pBI->biHeight = 100;
             pBI->biPlanes = 1;
             pBI->biBitCount = 32;
             pBI->biCompression = BI_RGB;
             pBI->biSizeImage = 0;
             pBI->biXPelsPerMeter = 0;
             pBI->biYPelsPerMeter = 0;
             pBI->biClrUsed = 0;
             pBI->biClrImportant = 0;

             m_pBits = new long[100*100*4];
      }
```

```
    ~CDisplay(){
        if (m_pBMI)
            delete m_pBMI;

        if (m_pBits)
            delete [] m_pBits;
    }

// IUnknown
    STDMETHOD(QueryInterface)(REFIID riid, void **ppv){

        *ppv = NULL;

        if (riid == IID_IUnknown || riid == IID_IDisplay){
            *ppv = this;
            this->AddRef();
        }

        if (!ppv)
            return E_NOINTERFACE;

        return S_OK;
    }

    DWORD STDMETHODCALLTYPE AddRef(){
        return ++m_nRef;
    }

    DWORD STDMETHODCALLTYPE Release(){
        if (--m_nRef == 0)
            delete this;

        return m_nRef;
    }

// IDisplay
    STDMETHOD(PutPixel)(long x, long y, long color){

        m_pBits[GetIndex(x,y)] = color;

        return S_OK;
    }

    STDMETHOD(Repaint)(){
        AfxGetMainWnd()->Invalidate(FALSE);

        return S_OK;
    }

    BITMAPINFO *m_pBMI;
    long *m_pBits;

private:
    long GetIndex(int x, int y){
        BITMAPINFOHEADER *pBI = (BITMAPINFOHEADER *) m_pBMI;

        long yoffset = pBI->biWidth * y;
        long xoffset = x;

        return xoffset + yoffset;
    }

    DWORD m_nRef;
};

#endif
```

10.1.3 Fractal Server Code

Below is the code that wraps the *CMandel* object. It creates a dynamically linked library, and appropriate information must be inserted into the registry before it can run. The ActiveX Template Library project automatically does this. The *CMandel::calculate()* not calculates the fractal points, and transmits the data to the client. This also makes use of the *complexPart* class described in the CORBA implementation section.

10.1.3.1 Connect.cpp

```
// Connect.cpp : Implementation of DLL Exports.

#include "preconn.h"
#include "connres.h"
#include "initguid.h"
#include "Connect.h"
#include "Mandel.h"

#define IID_DEFINED
#include "Connect_i.c"

CComModule _Module;

BEGIN_OBJECT_MAP(ObjectMap)
	OBJECT_ENTRY(CLSID_CoMandel, CMandel)
END_OBJECT_MAP()

/////////////////////////////////////////////////////////////////////////////
// DLL Entry Point

extern "C"
BOOL WINAPI DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID
/*lpReserved*/)
{
	if (dwReason == DLL_PROCESS_ATTACH)
	{
		_Module.Init(ObjectMap, hInstance);
		DisableThreadLibraryCalls(hInstance);
	}
	else if (dwReason == DLL_PROCESS_DETACH)
		_Module.Term();
	return TRUE;    // ok
}

/////////////////////////////////////////////////////////////////////////////
// Used to determine whether the DLL can be unloaded by OLE

STDAPI DllCanUnloadNow(void)
{
	return (_Module.GetLockCount()==0) ? S_OK : S_FALSE;
}

/////////////////////////////////////////////////////////////////////////////
// Returns a class factory to create an object of the requested type

STDAPI DllGetClassObject(REFCLSID rclsid, REFIID riid, LPVOID* ppv)
{
	return _Module.GetClassObject(rclsid, riid, ppv);
}

/////////////////////////////////////////////////////////////////////////////
// DllRegisterServer - Adds entries to the system registry

STDAPI DllRegisterServer(void)
```

```
{
      // registers object, typelib and all interfaces in typelib
      return _Module.RegisterServer(TRUE);
}

/////////////////////////////////////////////////////////////////////////////
// DllUnregisterServer - Removes entries from the system registry

STDAPI DllUnregisterServer(void)
{
      _Module.UnregisterServer();
      return S_OK;
}
```

10.1.3.2 Mandel.cpp

```
// Mandel.cpp : Implementation of CConnectApp and DLL registration.

#include "preconn.h"
#include "Connect.h"
#include "Mandel.h"
#include "complexpart.h"
#include <stdio.h>

/////////////////////////////////////////////////////////////////////////////
//

STDMETHODIMP CMandel::InterfaceSupportsErrorInfo(REFIID riid)
{
  if (riid == IID_IMandel)
    return S_OK;
  return S_FALSE;
}

STDMETHODIMP CMandel::calculate(double fpointX,
            double fpointY,
            double fpointWidth,
            double fpointHeight,
            long nXcorner,
            long nYcorner,
            long nWidth,
            long nHeight){
  HRESULT hr = S_OK;

  IConnectionPointImpl<CMandel, &IID_IDisplay, CComDynamicUnkArray>* p = this;

  if (!ReadMapFile("default.map"))
  {
    MessageBox(NULL, "Could not read color map file", "Error", MB_OK);
    return NOERROR;
  }

  long *FrameBuffer = new long[nWidth * nHeight];
  double ftempX = fpointX, ftempY = fpointY,
         fXinterval = fpointWidth / nWidth,
         fYinterval = fpointHeight / nHeight;
  long lIterations;

  for (long y = 0; y < nHeight; ++y, ftempY += fYinterval){
    ftempX = fpointX;
    for (long x = 0; x < nWidth; ++x, ftempX += fXinterval){
      lIterations = MandelCalc(ftempX, ftempY);
      FrameBuffer[x + y * nWidth] = m_ColorSpace[lIterations % 256];
      {
        Lock();
        IUnknown** pp = p->m_vec.begin();
        while (pp < p->m_vec.end() && hr == S_OK)
```

```
          {
            if (*pp != NULL)
            {
              IDisplay* pIDisplay = (IDisplay*)*pp;
              hr = pIDisplay->PutPixel(x + nXcorner,
                                       y + nYcorner, FrameBuffer[x + y * nWidth]);
            }
            pp++;
          }
          Unlock();

      }
    }
    if (y % 2)
    {
      Lock();
      IUnknown** pp = p->m_vec.begin();
      while (pp < p->m_vec.end() && hr == S_OK)
      {
        if (*pp != NULL)
        {
          IDisplay* pIDisplay = (IDisplay*)*pp;
          hr = pIDisplay->Repaint();
        }
        pp++;
      }
      Unlock();
    }
  }

  Lock();
  IUnknown** pp = p->m_vec.begin();
  while (pp < p->m_vec.end() && hr == S_OK)
  {
    if (*pp != NULL)
    {
      IDisplay* pIDisplay = (IDisplay*)*pp;
      hr = pIDisplay->Repaint();
    }
    pp++;
  }
  Unlock();

  return hr;
}

long CMandel::MandelCalc(double x, double y){
        int i;

        complexPart<double> c(x,y);
        complexPart<double> z(0, 0);

        // Loop and see how many iterations it takes z
        // to go out of bounds, under m_lIterations
        for (i = 1; i < 1000; i++){
                z = z * z + c;
                if (norm(z) >= 4)
                        return i;
        }

        return 0;
}

int CMandel::ReadMapFile(char *name){
        FILE *file;

        file = fopen(name, "ra");
        if (file){
              for (int i = 0; i < 256; ++i){
                    char red, green, blue;

                    fscanf(file, "%d%d%d\n", &red, &green, &blue);

                    m_ColorSpace[i] = long((long(long(blue) << 16) +
                                                  long(long(green) << 8) +
                                                  long(red)));
              }
        }
        else
        {
              return 0;
        }
        fclose(file);
        return 1;
}
```

10.1.4 Display Server Code

Below is the code that wraps the *CDisplay* object as well as the *CMandel* client. The code that works with the client is in the *CDisplayconnDoc::OnNewFractal()* method. This application is an MFC application. The implementation for the object is actually in the *CDisplay.h* file, as inline methods.

10.1.4.1 displayconnDoc.h

```
// displayconn.h : main header file for the DISPLAYCONN application
//

#ifndef __AFXWIN_H__
      #error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h"       // main symbols

/////////////////////////////////////////////////////////////////////////////
// CDisplayconnApp:
// See displayconn.cpp for the implementation of this class
//

class CDisplayconnApp : public CWinApp
{
public:
      CDisplayconnApp();
      ~CDisplayconnApp();

// Overrides
      // ClassWizard generated virtual function overrides
      //{{AFX_VIRTUAL(CDisplayconnApp)
      public:
      virtual BOOL InitInstance();
      //}}AFX_VIRTUAL

// Implementation

      //{{AFX_MSG(CDisplayconnApp)
      afx_msg void OnAppAbout();
            // NOTE - the ClassWizard will add and remove member functions
here.
            //    DO NOT EDIT what you see in these blocks of generated code
!
      //}}AFX_MSG
```

```
	DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////////
```

10.1.4.2 displayconnDoc.cpp

```
// displayconnDoc.cpp : implementation of the CDisplayconnDoc class
//

#include "stdafx.h"
#include "displayconn.h"
#include "displayconnDoc.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CDisplayconnDoc

IMPLEMENT_DYNCREATE(CDisplayconnDoc, CDocument)

BEGIN_MESSAGE_MAP(CDisplayconnDoc, CDocument)
	//{{AFX_MSG_MAP(CDisplayconnDoc)
		// NOTE - the ClassWizard will add and remove mapping macros
here.
		//    DO NOT EDIT what you see in these blocks of generated code!
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()

BOOL CDisplayconnDoc::OnNewDocument()
{
	return TRUE;
}

/////////////////////////////////////////////////////////////////////////////
// CDisplayconnDoc serialization

void CDisplayconnDoc::Serialize(CArchive& ar)
{
	if (ar.IsStoring())
	{
		// TODO: add storing code here
	}
	else
	{
		// TODO: add loading code here
	}
}

/////////////////////////////////////////////////////////////////////////////
// CDisplayconnDoc diagnostics

#ifdef _DEBUG
void CDisplayconnDoc::AssertValid() const
{
	CDocument::AssertValid();
}

void CDisplayconnDoc::Dump(CDumpContext& dc) const
{
	CDocument::Dump(dc);
}
```

```
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// Construction/Destruction
CDisplayconnDoc::CDisplayconnDoc()
{
	m_fStartX = -2;
	m_fStartY = -1.5;
	m_fpointWidth = 3;
	m_fpointHeight = 3;
	m_nWidth = 150;
	m_nHeight = 150;
	m_pDisplayImpl = new CDisplay;
	HRESULT hr = CoCreateInstance(CLSID_CoMandel,
					NULL, CLSCTX_ALL,
					IID_IMandel, (void**)&m_pM);
	_ASSERTE(m_pM != NULL);

	m_dwAdvise = 0;
}

CDisplayconnDoc::~CDisplayconnDoc()
{
	if (m_pDisplayImpl){
		delete m_pDisplayImpl;
	}

//	if (m_dwAdvise != 0)
//		AtlUnadvise(m_pM, IID_IDisplay, m_dwAdvise);
}

/////////////////////////////////////////////////////////////////////////////
// CDisplayconnDoc commands

int CDisplayconnDoc::GetWidth() const
{
	BITMAPINFOHEADER *pBI = (BITMAPINFOHEADER *)m_pDisplayImpl->m_pBMI;

	return pBI->biWidth;
}

int CDisplayconnDoc::GetHeight() const
{
	BITMAPINFOHEADER *pBI = (BITMAPINFOHEADER *)m_pDisplayImpl->m_pBMI;

	return abs(pBI->biHeight);
}

long *CDisplayconnDoc::GetBitsAddress() const
{
	return m_pDisplayImpl->m_pBits;
}

BITMAPINFO *CDisplayconnDoc::GetBitmapInfo() const
{
	return m_pDisplayImpl->m_pBMI;
}

void CDisplayconnDoc::OnNewFractal(){

	HRESULT hr;

	if (m_dwAdvise != 0)
		AtlUnadvise(m_pM, IID_IDisplay, m_dwAdvise);

	if (m_pDisplayImpl)
		delete m_pDisplayImpl;
```

```
        m_pDisplayImpl = new CDisplay;

        hr = AtlAdvise(m_pM, m_pDisplayImpl, IID_IDisplay, &m_dwAdvise);
        m_pM->calculate(-2,-1.5,3,3,0,0,100,100);

        return;
}
```

10.2 DCOM Implementation Conclusion

Working with COM for the first time was a lot more difficult and time consuming than working with CORBA. The documentation available was all the documentation that came with Visual C++ 4.0 as well as the Microsoft Developer's Library of January 1997.

The ActiveX Template Library, failed as documented. At least three projects were created using Microsoft's ATL wizard, following their documentation, following their sample on connection points, and they all failed to work. The final project was a modification of their sample project.

One reason why it was so difficult was because of the complexity in DCOM; There are many ways to do the same thing. While this is allows great flexibility, it greatly increases the chances of bugs. Even Microsoft's integrated tools do not do enough of a job in determining the source of run-time errors. This makes the DCOM learning curve very steep.

11. Alternatives

Besides the technologies described in detail, there are other technologies that others are working on. There are technologies out there that are either in experimental development, or not used widely. The following are some of the more popular companies and their projects.

11.1 The Open Group

Perhaps the most widely known of these is the Open Software Foundation's (now called the Open Group) Distributed Computing Environment (DCE). It is a suite of software and services that enable scaleable distributed solutions on heterogeneous networks. It has been around much longer than ActiveX or CORBA, so it has many vendors and many adopters.

DCE services cover much of what DCOM and CORBA provide. The difference between DCE and the other technologies presented in this thesis is that DCE does not inherently support objects. While there are proponents who claim that DCE can be adapted to do distributed object computing, The Open Group is not about to create a competitor to Java's Distributed Object Model, DCOM, and CORBA. However, the OMG has recognized the value of DCE's RPC, security services, and directory services and have designed CORBA so that it can integrate with these aspects.

Adding objects to DCE at this point would probably be a wasted effort since it is already behind the other technologies (except Java's Distributed Object Model) and Microsoft has already specified an Object RPC. The Open Group is currently researching uses of CORBA, ActiveX, and Java instead of competing with them.

11.2 Xerox

Xerox's Palo Alto Research Center has come up with an object-oriented solution called Inter-Language Unification (ILU). The goal of this project is to allow interaction between different program modules. Each module is encapsulated by an object. These objects learn how to communicate with each other using Xerox's Interface Specification Language (ISL) or CORBA IDL.

Even though ILU allows CORBA IDL specification, it is not an effective CORBA implementation. It does not provide CORBA-specified services, even though they provide their own services that do the same thing as the CORBA services. There are many differences between this infrastructure and CORBA. It also does not have DII, DSI, an interface repository, or an implementation repository.

However, ILU addresses certain topic that are not covered by CORBA:

1. Garbage Collection

   One major advantage that it provides over CORBA is automatic garbage collection. ILU objects can be tagged as collectible, so the server keeps of who uses it. It uses timeouts to garbage collect in the face of client failure.

2. UNICODE

   Currently, CORBA does not address the issue of Unicode strings, like DCOM does. The `string` in CORBA's IDL is standard 8-bit, zero-terminated string.

3. 64-bit Architectures
   CORBA's data types currently do not address 64 bit architectures and the new data types that are associated with it. However, including this into CORBA should not be a very difficult.

Currently, ILU works with C, C++, Python, Java, Common Lisp, and Modula-3.

11.3 NeXT

NeXT has created a set of technologies that run on the WWW, using a Java-based client code and their own server. This technology is called WebObjects. WebObjects was first released in 1995, and it overlaps with much of what DCOM and CORBA have been doing. However, WebObjects applications can integrate with CORBA, Windows, mainframe and database systems and runs on many platforms.

WebObjects allows dynamic web applications and is the middleware between HTML and native data. It specifically targets performance, UI generation, and state management. To improve performance, WebObjects allows distribution of services onto other servers. To facilitate UI generation, WebObjects lets developers work with components when dynamically creating web pages. This lets developers create good user interfaces easily, without worrying about how the HTML will look (and especially with dynamic web pages, it is hard to see the end results with traditional methods). State management is an important part of WWW transactions. Developers used to do it with cookies and CGI-scripted web pages, but WebObjects provides an extensive infrastructure to ease transactions.

11.4 Ring Server Project

The government sponsored Ring Server Project in Japan is researching methods that will enable a large scale software archive site. In the process, it has come up with a Java RMI-like infrastructure called HORB (not CORBA related). It runs on any Java VM, and unlike RMI, it can run on top of JDK 1.0.

The whole project is open and anyone can download and look at the source code. Similar to RMI, it requires no IDL files. However, it does use client and server stubs (generated from the Java interfaces themselves). HORB has extra features such as distributed garbage collection and remote object creation. This latter feature is something unable to be done on any system currently. It lets a client ask a server to create a new object without *a priori* knowledge by the server. This can add a new dimension to dynamic networks, and something that only Java can do because of its portable bytecodes.

HORB, however, is not compatible with CORBA or Java RMI. A user who needs this functionality can write a bridge to connect the technologies.

11.5 Information Technology Promotion Agency (Japan)

The Information Technology Promotion Agency (IPA) of Japan has created an infrastructure for distributed objects, the Open Fundamental Software Technology Project. The phrase "fundamental software" is middleware.

There is a research area of this project called OZ++ which provides mechanisms that allow reuse and object sharing. In essence, this project is doing what the OMG is doing. Not much research is publicly available, but it seems that this project does not have as much support as CORBA, and is not truly heterogeneous. They claim to be heterogeneous in that objects can run on different hardware and operating systems, but each object must be written in a new language they developed, the OZ++ language. It is based on C, but excludes pointers, structs, and other mechanism and replaces them with classes. It seems that this project can not be effective in reusing existing code or integrating with legacy systems.

12. Conclusion

Now the question is, "what's the bottom line?" Which technology is the best to use? The answer to this question really depends on the needs and existing infrastructure. In fact, object-oriented technologies are not the only options, especially in projects that need mature technologies right now.

12.1 Standard Object Technologies

DCOM, Java RMI, or CORBA? If the system will run completely on Windows machines, then DCOM might be a good solution. If the system will run completely on Java machines, then Java RMI is an option. However, any heterogeneous will probably fare better with CORBA.

While DCOM has good roots in COM, the network integration is still immature. In addition Windows NT servers with DCOM enabled are vulnerable to denial of service attacks because of a bug. Thus, any DCOM infrastructure *must* stay within a firewall. Assuming that in the future this bug does not exist, then DCOM as an infrastructure is a lot more feasible since it contains packet level encryption.

Development tools for DCOM are the best out of all these technologies, and probably will be for a long time. This is because of Microsoft's commitment to DCOM as well as its huge market share of compilers and desktop operating systems. However, DCOM support outside of Windows should be taken with a grain of salt, and tools on those platforms will definitely not be mature.

Sun is avidly pushing a "100% Java" campaign, but currently the world is not. Programmers who wish to use Java RMI for a project should consider:

1. Is the project small enough to be done effectively with Java RMI?
2. Is there no way this project will need to be integrated with modules in other languages in the future?

While 1 is easy to answer, 2 is not. Who knows what someone will do with your code in 10 years? Locking into Java RMI is probably not the best thing to do, unless the nature of the project is that it will not continue for too long.

CORBA is the most mature technology of all these, with the advantage that it can work with legacy applications and heterogeneous networks. One problem is that development tools for CORBA are not very mature. Some ORBs do not bundle tools that integrate with existing compilers (although older ORBs like ORB Plus and Orbix do bundle tools that work with Visual C++). Currently, however, few vendors have implemented the Security Service, so mass deployment over the Internet might not be as secure, especially if the project uses the public IIOP protocol on the wire. By the end of this year, though, many vendors will have implemented the service.

CORBA at this point looks like the best choice for distributed object technology deployment. The other technologies can not really beat out CORBA.

12.2 Downloadable Technologies

The two downloadable object technologies are JavaBeans and ActiveX. Under a good system administrator, ActiveX could be a secure technology to use. As long as web browsers were not allowed to download components outside the company, then there would be nothing to worry about.

However, Java Beans allows the same type of flexibility that ActiveX controls have, and in addition, they do not have to ask the user to trust them. They can also run on heterogeneous networks, where each platform has a Java VM. It is also possible for a trusted applet to call native functions of its host operating system, although this would remove its advantage of platform independence.

Tools for developing ActiveX controls are more mature than those for developing Java Beans. Like with DCOM, Microsoft is putting its full weight in promoting this technology, leveraging its huge market share. However, users will have to take into consideration the fact that not everyone in the world runs Microsoft software. JavaBeans is the technology the can reach the broadest possible audience.

12.3 Implementation Comparisons

From the sample implementation experiences detailed in Sections 9 and 10, CORBA is probably the easiest architecture to program with and learn; using C++ to use CORBA is not very different from straight C++. However, CORBA is not as integrated into Windows as DCOM is, and also requires expensive licenses. Deciding on a distributed object technology is not easy; many factors are involved, including programming issues and business issues.

12.4 Bottom Line

Who knows whether Microsoft or its competitors will win the battle of the technologies? To protect one's investment in object technology, it is possible to write an "ORB isolation layer" that abstracts away the infrastructure from the application. This idea is further expounded on in [Roy]. The ultimate choice of the chosen technology will depend on the actual project being used, and whether or not the user is willing to lock into Microsoft or not. Hopefully this thesis has given a few insights on what factors to look for in deciding on an object technology infrastructure.

13. References

[Borland] "Borland Licenses Visigenic Object Request Broker Technology". *http://www.visigenic.com/news/bjb.html.* March 4, 1997.

[Brockschmidt] Brockschmidt, Kraig. "What OLE is Really About". *http://www.microsoft.com/oledev/olecom/aboutole.htm.* Microsoft Corporation, July, 1996.

[Brockschmidt2] Brockschmidt, Kraig. *Inside OLE.* Microsoft Press, Redmond: 1995.

[Chappell] Chappell, Dave. "DCE and Objects". *http://www.chappellassoc.com/DCEobj.htm, http://www.opengroup.org/tech/dce/3rd-party/ChapRpt1.html.* 1996.

[Cline] Cline, Marshall. "C++ FAQ Lite". *http://www.cerfnet.com/~mpcline/c++-faq-lite/.* 1996.

[CORBANet] "CORBAnet - The ORB Interoperability Showcase". *http://corbanet.dstc.edu.au/.*

[CORBAWeb] Merle, Philippe; Gransart, Christophe; Geib, Jean-Marc. "CorbaWeb: A Navigator For CORBA Objects". *Dr. Dobb's Sourcebook* January/February 1997. p. 7.

[CSVB] "VB5.0: No Longer So 'Basic'". *Client/Server Computing.* March 1997. p15.

[DCOMBus] "A Business Overview". *Microsoft Windows NT Server White Paper. http://www.microsoft.com/ntserver/dcombus.exe.* 1996.

[DCOMSol] "DCOM Solutions in Action". *Microsoft Windows NT Server White Paper. http://www.microsoft.com/ntserver/dcomsol.exe.* 1996.

[DCOMTec] "DCOM Technical Overview". *Microsoft Windows NT Server White Paper. http://www.microsoft.com/ntserver/dcomtec.exe.* 1996.

[DCOMHB] "DCOM—Cariplo Home Banking Over The Internet". *Microsoft Windows NT Server White Paper. http://www.microsoft.com/ntserver/dcomhb.exe.* 1996.

[DCOMSec] "Microsoft® Windows NT® Distributed Security Services: Secure Networking using Windows NT Server Distributed Services Technology Preview" . *Microsoft Windows NT Server White Paper. http://www.microsoft.com/ntserver/dcomhb.exe* 1996

[DOMISImpl] Brando, T. J., *DOMIS Implementation of CTAPS Functionality Using Orbix*, MP 94B-287, The MITRE Corporation, Bedford, MA, December 1994.

[DOMISFY94] Brando, T. J., M. P. Chase, I. M. Kyratzoglou, D. A. Ondzes, M. J. Prelle, A. S. Rosenthal, A. L. Schafer, and A. M. Tallant, *Distributed Object Management Integration System (DOMIS) FY94 Final Report*, MP 95B-320, The MITRE Corporation, Bedford, MA, September 1995. (This is the public release version of a report that was originally published in October of 1994.)

[DOD] *Trusted Computer System Evaluation Criteria*. US Department of Defense. December, 1985.

[HM] Harmon, Paul; Morrissey, William. *The Object Technology Casebook*. John Wiley and Son, Inc. New York: 1996.

[Intergalactic] Resnick, Ron I. "Intergalactic Distributed Objects". *Dr. Dobb's Sourcebook* January/February 1997. p. 35.

[InterLanguage] Genereaux, Tom. "The InterLanguage Unification System". *Dr. Dobb's Sourcebook* January/February 1997. p. 41.

[JavaRMI] *Java™ Remote Method Invocation Specification. http://chatsubo.javasoft.com/current/*. Rev 1.4. Sun Microsystems. 1997.

[JavaBean] *JavaBeans™. http://java.sun.com/beans*. Sun Microsystems. 1997.

[LT] Leveson, Nancy G. and Turner, Clark S. "An investigation of the Therac-25 accidents". *Computer* Vol. 26, No. 7. July, 1993, pp. 18-41.

[McLain] McLain, Fred. "ActiveX or how to put nuclear bombs in web pages". *http://www.halcyon.com/mclain/ActiveX/*

[MSA] "Microsoft Security Advisor". *http://www.microsoft.com/security/*.

[NetReady] O'Brien, Timothy; Heise, Douglas. "Java Beans, ActiveX, Which Path to Choose?" *NetReady Advisor*. Winter 1997. p. 16. SIGS Publications

[Netscape] "Netscape Expands Use Of Visigenic's ORB Technology With New Agreement For Netscape Enterprise Server 3.0". *http://www.visigenic.com/news/Netscape97.html*. February 5, 1997.

[Newman] Newman, David S. "Managing CORBA Method Systems". *Distributed Object Computing*. p. 43. Vol 1. Issue 1. February, 1997.

[Novell] "Novell Licenses Visigenic's Leading Object Request Broker Technology". *http://www.visigenic.com/news/novell397.html.* March 25, 1997.

[Oracle] "Oracle and Visigenic Join Forces to Deliver Best of Breed Object Technology, Java and Open Standards for Network Computing Architecture". *http://www.visigenic.com/news/Oracle97.html.* February 5, 1997.

[OHE] Orfali, Robert; Harkey, Dan; and Edwards, Jeri. *The Essential Distributed Objects Survival Guide.* John Wiley and Son, Inc. New York: 1996.

[OHE2] Orfali, Robert; Harkey, Dan; and Edwards, Jeri. *Instant CORBA.* John Wiley and Sons, Inc. New York: 1997.

[OpenGroup] "ActiveX ATO Proposals - 1997". *http://www.osf.org/RI/ATO/actX/index.htm.*

[OFSTP] "Open Fundamental Software Technology Project - Japan". *http://www.ipa.go.jp/OFSTP/home.html*

[Rauch] Rauch, Stephen. "Talk to Any Database the COM Way Using OLE DB". *http://www.microsoft.com/oledb/prodinfo/msjrauch/rauch.htm*

[Roy] Roy, Mark; Ewald, Alan. "Defining & Building". *Distributed Object Computing.* February 1997, p. 53.

[SGI] Silicon Graphics Press Release. *http://www.iona.com/Orbix/Customers/SiliconGraphics.html.*

[Siegel] Siegel, Jon. *CORBA Fundamentals and Programming.* John Wiley and Sons, Inc. New York: 1996.

[Swartz] Swartz, John. "Simulating the Denver Airport Automated Baggage System". *Dr. Dobbs Journal.* #261. January 1997, pp. 56-62.

[Taivalsaari] Taivalsaari, Antero. "On the Notion of Inheritance". *ACM Computing Surveys* Vol. 28, No. 3. September 1996, pp. 438-479.

[UKIm] "UK Immigration Service Relies On DAIS For Critical Document Management" *http://www.icl.com/products/dais/cshosip.html*

[UNO] "Interoperability and the CORBA Specification". *http://www.mitre.org/research/domis/reports/UNO.html*

[VisiCPPPG] *VisiBroker for C++ Programmer's Guide Version 2.0.* Visigenic Software: 1996.