On Designing and Developing CORBA Based Applications

Yi Li

A Major Report

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal Quebec Canada

April 2004

Your file Votre référence
ISBN: 0-612-91065-2
Our file Notre référence
ISBN: 0-612-91065-2

Abstract

On Designing and Developing CORBA Based Applications

Yi Li

Common Object Request Broker (CORBA) is a mechanism to create, deploy, and manage objects in a distributed environment. These objects are generally referred to as distributed components or CORBA components.

CORBA is an open standard that addresses object orientation, distribution transparency, hardware independence, operating system independence, language independence and vendor independence.

This report first highlights: What is CORBA? Why is CORBA needed? How is CORBA used for developing applications? Second, this report shows how the CORBA architecture is used to develop distributed object-based game applications.

Then, a CORBA-based application of gomoku game is given. The game is implemented with VisiBroker 4.0 (a CORBA 2.3 specification compliance product) and C++ language.

The game can be run on any platform for which a CORBA product (for instance, VisiBroker or Orbix) is installed. The approach used in this report can be easily used for other games in the distributed environment.

Acknowledgments

I would like to sincerely thank my supervisor Dr. Peter Grogono for his patient and valuable suggestions and instructions. Without his encouragement this work would not have been possible.

I would also like to thank Dr. J. William Atwood for his valuable instructions and reading my major report carefully.

I would also like to thank Ms. Halina Monkiewicz, the Graduate Program Advisor in Computer Science, for her enthusiastic help.

I would also like to thank all the people who help me in various ways in my doing this report.

Last but not least, I would like to thank my lovely wife and son for their understanding and continuous support.

Table of Contents

Chapter 1 Introduction

Modern programming languages employ the object paradigm to structure computation within a single operating system process. The next logical step is to distribute a computation over multiple processes on one single or even on different machines. Because object orientation has proven to be an adequate means for developing and maintaining large scale applications, it seems reasonable to apply the object paradigm to distributed computation as well: objects are distributed over the machines within a networked environment and communicate with each other.

As a fact of life the computers within a networked environment differ in hardware architectures, operating system software, and the programming languages used to implement the objects. This is called a *heterogeneous* distributed environment. To allow communication between objects in such an environment one needs a rather complex piece of software called a *middleware* platform. Middleware refers to software that aids in the creation of distributed applications. Middleware generally provides a framework for developing distributed applications, shielding the programmer from lower level details, providing run-time application support features.

The *Common Object Request Broker Architecture* (CORBA) is a specification of such a middleware platform by the *Object Management Group* (OMG).

This report first highlights: What is CORBA? Why is CORBA needed? How is CORBA used for developing applications?

Second, this report shows how the CORBA architecture is used to develop distributed object-based game applications.

Then, a CORBA-based application of *gomoku* game is given. The game is implemented with VisiBroker 4.0 (a CORBA 2.3 specification compliance product) and C++ language.

The *object-oriented design* (OOD) methodology is used fully in designing and implementing the game application.

The game can be run on any platform on which a CORBA product (for instance, VisiBroker or Orbix) is installed.

Chapter 2 Why CORBA

CORBA provides a flexible communication and activation substrate for distributed heterogeneous object-oriented computing environments. When designing and implementing distributed applications, CORBA certainly is not the only choice. Other mechanisms exist by which such applications can be built. We will briefly explore some of the alternatives and see how they compare to CORBA.

2.1 The Object Management Group (OMG)

To address the problems of distributed computing, the *Object Management Group* (OMG) [6] was established in 1989 with eight original members, and is now a 760 plus member organization whose charter is to "provide a common architectural framework for object-oriented applications based on widely available interface specifications." That is a rather tall order, but the OMG achieves its goals with the establishment of the *Object Management Architecture* (OMA), of which CORBA is a part. This set of standards delivers the common architectural framework on which applications are built. Very briefly, the OMA consists of the *Object Request Broker* (ORB) function, object services (known as *CORBAServices*), common facilities (known as *CORBAfacilities*), domain interfaces, and application objects. CORBA's role in the OMA is to implement the Object Request Broker function.

2.2 CORBA Architecture Overview

CORBA is a mechanism to create, deploy, and manage objects in a distributed environment. These objects are generally referred to as distributed components or *CORBA components*.

CORBA is also an open standard that addresses object orientation, distribution transparency, hardware independence, operating system independence, language independence and vendor independence.

CORBA is an object-oriented architecture. CORBA objects exhibit many features and traits of other object-oriented systems, including interface inheritance and polymorphism. Although CORBA maps particularly well to object-oriented languages, such as C++ and Java, CORBA provides the object-oriented capability even when used with nonobject-oriented languages such as C and Cobol. Interface inheritance is a concept that should be familiar to C++ and Java developers. In the contrasting implementation inheritance, an implementation unit (usually a class) can be derived from another. By comparison, interface inheritance allows an interface to be derived from another. Even though interfaces can be related through inheritance, the implementations for those interfaces need not be.

2.3 Why distributed computing

Today's business is driven by computerized information. Information must be accessible through networks. Typical organizations have the following characteristics:

- Organizations are distributed.
- Business relationships are distributed.
- Computing resources are distributed.
- Enterprise computing needs to span organization and enterprise.
- Network programming is time consuming.
- Networks are almost always heterogeneous.

Distributed computing has all of these characteristics and is therefore suitable for modern organizations. In a distributed computing environment, geographical limitations are reduced and distant individuals can communicate and collaborate more effectively; computing resources (in terms of both software and hardware) are shared on the networks and access to shared information can be greatly improved.

Distributed object computing extends an object-oriented programming system by allowing objects to be distributed across a heterogeneous network, so that each of these distributed object components interoperate as a unified whole. These objects may be distributed on different computers throughout a network, living within their own address space outside of an application, and yet appear as though they were local to an application.

2.4 Socket Programming

In the most modern systems, communication between machines, and sometimes between processes in the same machine, is done through the use of *sockets*. Simply put, a socket is a channel through which applications can connect with each other and communicate. The most straightforward way to communicate between application components is to use sockets directly (this is known as *socket programming*). That is, the developers write data to and read data from sockets.

The *Application Programming Interface* (API) for socket programming is rather low-level. So, the overhead associated with an application that communicates in this fashion is very low. Because the API is low-level, socket programming is not well suited to handling complex data types, especially when application components reside on different type of machines or are implemented in different programming languages. Whereas direct socket programming can result in very efficient applications, the approach is usually unsuitable for developing complex distributed applications.

Socket network programming mechanism also lacks type safe, portable and extensible interfaces.

2.5 Remote Procedure Call (RPC)

One rung on the ladder above socket programming is *remote procedure call* (RPC). RPC provides a function-oriented interface to socket-level communications. Using RPC, rather than directly manipulating the data that flows to and from a socket, the developer defines a function – much like those in a functional language such as C – and generates code that makes that function look like a normal function to the caller. Under the hood, the function actually uses sockets to communicate with a remote server, which executes the function and returns the result, again using sockets.

Because RPC provides a function-oriented interface, it is often much easier to use than raw socket programming. RPC is also powerful enough to be the basis for many *client / server* applications. Although there are various incompatible implementations of RPC protocol, a standard RPC protocol exists that is readily available for most platforms [5].

2.6 OSF Distributed Computing Environment (DCE)

The *Distributed Computing Environment* (DCE), a set of standards pioneered by the *Open Software Foundation* (OSF), includes a standard for RPC.

DCE provides a complete Distributed Computing Environment infrastructure. It provides security services to protect and control access to data, name services that make it easy to find distributed resources, and a highly scalable model for organizing widely scattered users, services, and data. DCE runs on all major computing platforms and is designed to support distributed applications in heterogeneous hardware and software environments.

Although the DCE standard has been around for some time, and was probably a good idea, it has never gained wide acceptance and exists today as little more than a historical curiosity [5].

2.7 Microsoft Distributed Component Object Model (DCOM)

Microsoft Distributed Component Object Model (DCOM), which is often called "*COM on the wire*", supports remote objects by running on a protocol called the *Object Remote Procedure Call* (ORPC). This ORPC layer is built on top of DCE's RPC and interacts with COM's run-time services. A DCOM server is a body of code that is capable of serving up objects of a particular type at runtime. Each DCOM server object can support *multiple interfaces,* each representing a different behavior of the object. A DCOM client calls into the exposed methods of a DCOM server by acquiring a pointer to one of the server object's interfaces. The client object then starts calling the server object's exposed methods through the acquired interface pointer as if the server object resided in the client's address space. As specified by COM, a server object's memory layout conforms to the C++ vtable layout. Since the COM specification is at the binary level it allows DCOM server components to be written in diverse programming languages such as C++, Java, Object Pascal (Delphi), Visual Basic and even COBOL. As long as a platform supports COM services, DCOM can be used on that platform.

The Distributed Component Object Model (DCOM) is Microsoft's entry into the distributed computing foray, offers capabilities similar to CORBA. DCOM is a relatively robust object model that enjoys particularly good support on Microsoft operating systems because it is integrated with Windows 95, Windows NT and the later versions.

Microsoft has, on numerous occasions, made it clear that DCOM is best supported on Windows systems. So developers with cross-platform interests in mind would be well advised to evaluate the capabilities of DCOM on their platform(s) of interest before committing to the use of this technology. However, for the development of Windows-only applications, it is difficult to imagine a distributed computing framework that better integrates with the Windows operating systems.

2.8 Java Remote Method Invocation (RMI)

The tour of exploring CORBA alternatives stops with *Java Remote Method Invocation* (RMI), a very CORBA like architecture with a few twists.

Java/RMI relies on a protocol called the *Java Remote Method Protocol* (JRMP). Java relies heavily on Java Object Serialization, which allows objects to be marshaled (or transmitted) as a stream. Since Java Object Serialization is specific to Java, both the Java/RMI server object and the client object have to be written in Java. Each Java/RMI Server object defines an interface that can be used to access the server object outside of the current *Java Virtual Machine* (JVM) and on another machine's JVM. The interface exposes a set of methods that are indicative of the services offered by the server object. For a client to locate a server object for the first time, RMI depends on a naming mechanism called an *RMIRegistry* that runs on the Server machine and holds information about available Server Objects. A Java/RMI client acquires an object reference to a Java/RMI server object by doing a lookup for a Server Object reference and invokes methods on the Server Object as if the Java/RMI server object resided in the client's address space. Java/RMI server objects are named using *Uniform Resource Locators* (URLs) and for a client to acquire a server object reference, it should specify the URL of the server object as you would with the URL to a HTML page. Since Java/RMI relies on Java, it can be used on diverse operating system platforms from mainframes to UNIX boxes to Windows machines to handheld devices as long as there is a Java Virtual Machine (JVM) implementation for that platform.

One advantage of RMI is that it supports the passing of objects by value, a feature not (currently) supported by CORBA. A disadvantage, however, is that RMI is a Java-only solution; that is, RMI clients and servers must be written in Java.

For all java applications, particularly those that benefit from the capability to pass object by value, RMI might be a good choice, but if there is a chance that the application will later need to interoperate with other applications written in other languages, CORBA is a better choice. Fortunately, full CORBA implementations already exist for Java, ensuring that Java applications interoperate with the rest of the CORBA world.

Chapter 3 CORBA Architecture

Having described the history and reasons for the existence of CORBA, we are ready to examine the CORBA architecture.

3.1 CORBA goals

The CORBA architecture has the features listed below:

- Location transparency: clients of objects do not need to know where the objects reside in the network.
- Separation of interface and implementation:
    - Enable object implementations to evolve independently for clients.
    - Enable programming language independence between client and object implementations
- *Marshalling* and *Unmarshalling* on-wire format: enable clients and objects in different operating systems to interoperate.
- Interoperability: allows communication between different vendors' products.

The classic CORBA architecture is as shown in Figure 1.

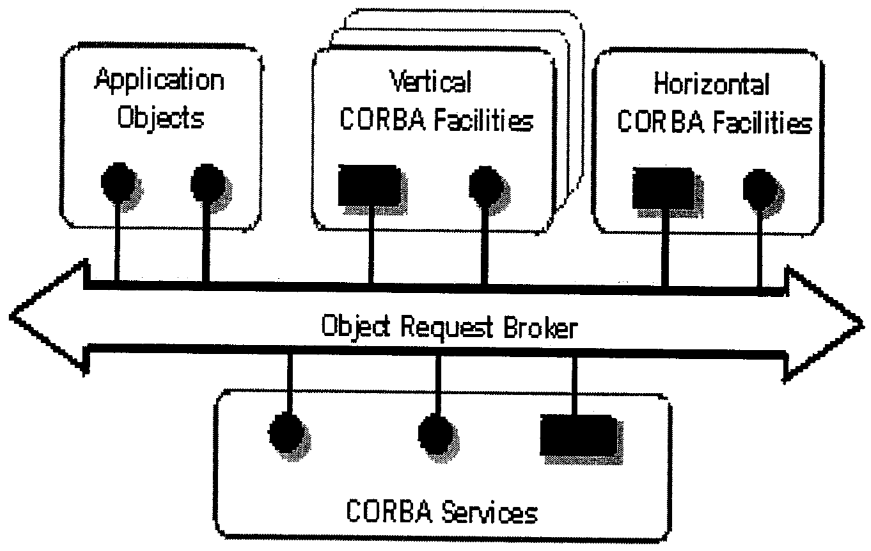


Figure 1

3.2 The Object Request Broker (ORB)

The Object Management Group (OMG) is an industry consortium that creates and publishes specifications for CORBA. Vendors implement the specifications to produce products known as Object Request Brokers (ORB). Fundamental to the Common Object Request Broker Architecture is the Object Request Broker (ORB). An ORB is a software component whose purpose is to facilitate communication between objects. It does so by providing a number of capabilities, one of which is to locate a remote object, given an object reference. When an application component wants to use a service provided by another component, it first must obtain an object reference for the object providing the service. After an object reference has been obtained, the component can call methods on that object, thus accessing the desired services provided by that object. (The developer of the client component knows at compile time which methods are available from a particular server object.) The primary responsibility of ORB is to resolve requests for object references, enabling application components to establish connectivity with each other.

ORB defines the middleware to carry out component collaboration in a distributed environment. ORB is also referred to as an object-bus.

ORB enables clients to communicate with a remote component in a distributed environment. In other words, ORB provides transparency of a component's location, activation, communication, and implementation. Thus ORB is essential for building and packaging distributed components. ORB encapsulates some aspects of the underlying OS and the networking layer [4].

Another service provided by the ORB is the marshaling of parameters and return values to and from remote method invocations.

After an application component has obtained a reference to an object whose services the component wants to use, that component can invoke methods of that object. Generally, these methods take parameters as input and return other parameters as output. Another responsibility of ORB is to receive the input parameters from the component that is calling the method and to marshal these parameters. What this means is that the ORB translates the parameters into a format that can be transmitted across the network to the remote object. (This is sometimes referred to as an on-the-wire format.) The ORB also unmarshals the returned parameters, converting them from the on-the-wire format into a format that the calling component understands.

The entire marshaling process takes place without any intervention from users whatsoever. A client application simply invokes the desired remote method – which has the appearance of being a local method, as far as the client is concerned – and the result is returned or an exception is raised, again, just as would happen with a local method. The entire process of marshaling input parameters, initiating the method invocation on the server, and unmashaling the return parameters are performed automatically and transparently by the ORB [5][6].

The process of marshalling and unmarshalling parameters is handled completely by the ORB, entirely transparent to both the client and the server. Because the entire processes is handled by the ORB, so the developers need not concern themselves with the details of the mechanism by which the parameters are marshaled and unmarshaled.

The sequences of ORB resolution of object requests including processes of marshalling and unmarshalling are shown in Figure 2.

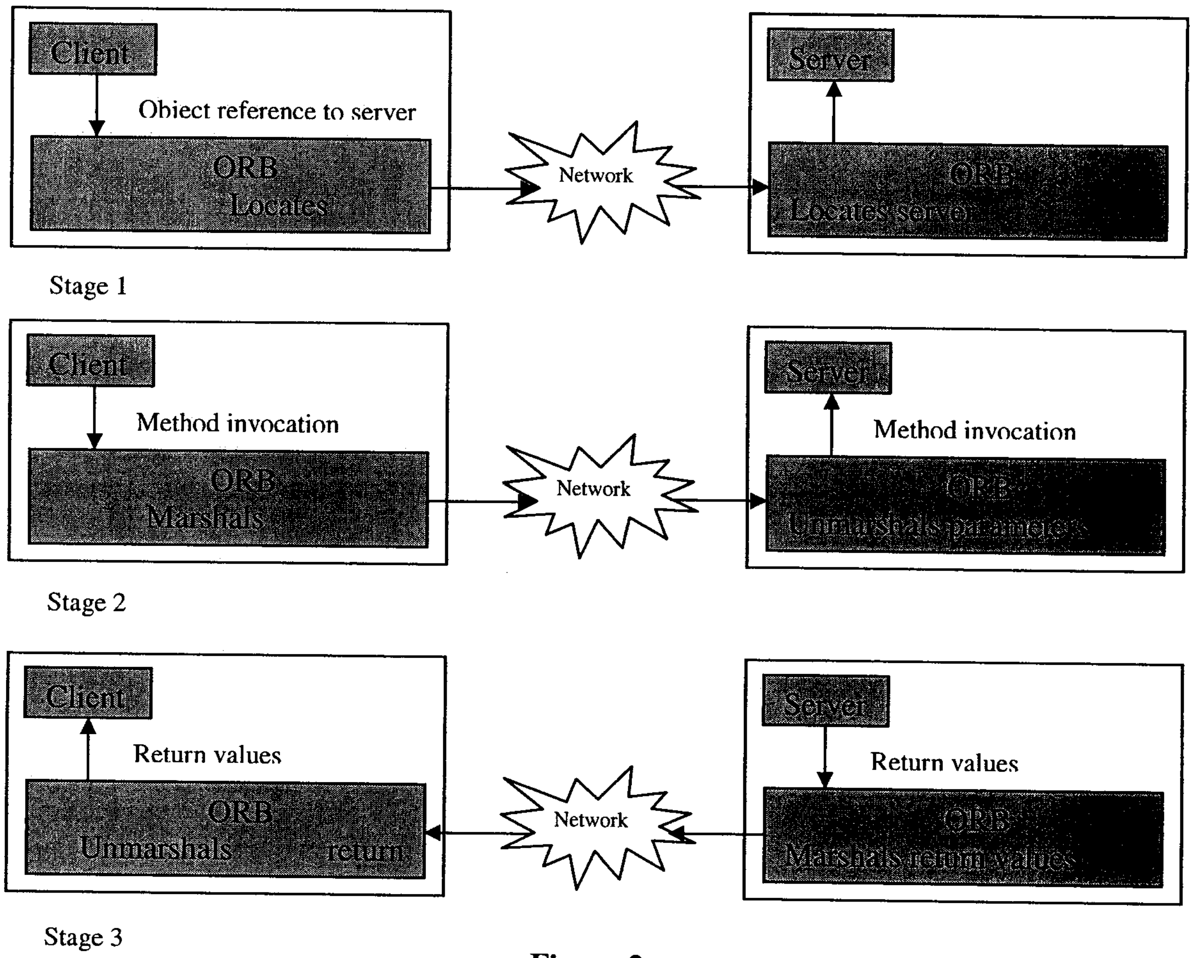


Figure 2

The use of ORB is one of the cornerstones of CORBA. ORB must run on both the client side and the server side. The ORB process on the client side is referred to as client-ORB, and the ORB process on the server side is referred as server-ORB.

The following summarizes the purpose of ORB and responsibilities that it carries out automatically and transparently:

1. Client-ORB locates a component implementation on behalf of the client when given an IOR (Interoperable Object Reference) by the client. A component can be located in a number of ways.
2. When a server is located, server-ORB prepares the server to receive the request. For example, server-ORB can launch the server if it is not running.
3. Client-ORB accepts the parameters from the client and marshals the parameters.
4. Server-ORB unmarshals the input parameters.
5. Server-ORB invokes the requested operation on the target component.
6. Server-ORB marshals the return parameters or exception.
7. Client-ORB unmarshals the return parameters or exception.

The major benefit offered by the ORB is its platform independent treatment of data; parameters can be converted on the fly between varying machine formats as they are marshaled and unmarshaled. So the communication between components is platform independent.

The Object Management Architecture (OMA) includes a provision for the ORB functionality; CORBA is the standard that implements this ORB capability [6]. We will soon see that the use of ORBs provides platform independence to distributed CORBA objects.

3.3 Interface Definition Language (IDL)

If the concept of the Object Request Broker is one cornerstone of the CORBA architecture, the *Interface Definition Language* (IDL) is another. IDL is a standard language used to define the interfaces used by CORBA objects [6]. As its name suggests, IDL is the language used to define interfaces between application components. That is, IDL can be only used to define interfaces, not implementations. It is not a procedural language. C++ programmers can think of IDL definitions as analogous to header files for classes; a header file typically does not contain any implementation of a class but rather describes that class's interface. Java programmers might liken IDL definitions to definitions of Java interfaces; again, only the interface is described, no implementation is provided with IDL.

IDL permits interfaces to objects to be defined independent of the implementation of the object. After defining an interface in IDL, the interface definition is used as input to an IDL compiler that produces output that can be compiled and linked with an object implementation and its clients.

Interfaces can be used either statically or dynamically. An interface is statically bound to an object when the name of the object it is accessing is known at compile time. In this case, the IDL compiler generates the necessary output to compile and link to the object at compile time. In addition, clients that need to discover an object at run time and construct a request dynamically can use the Dynamic Invocation Interface (DII). The DII is supported by an interface repository, which is defined as part of CORBA.

The IDL specification is responsible for ensuring that data is properly exchanged between dissimilar languages. For example, the IDL long type is a 32-bit signed integer quantity, which can map to a C++ long (depending on platform) or a Java int. It is the responsibility of the IDL specification and the IDL compilers that implement it to define such data types in a language independent way [6].

CORBA has been postured as being both language-neutral and independent of ORB and CORBA object implementations. The IDL language is part of the standard CORBA specification and is independent of any programming language. It achieves this language independence through the concept of a language mapping. The OMG has defined a number of standard language mappings for many popular languages, including C, C++ COBOL, Java, and Smalltalk. Mappings for other languages exist well; these mappings are either nondstandard or are in the process of being standardized by the OMG.

Language independence is a very important feature of the CORBA architecture. Because CORBA does not dictate a particular language to use, it gives application developers the freedom to choose the language that best suits the needs of their applications. Taking this freedom one step further, developers can also choose multiple languages for various components for an application. For instance, the client components of an application might be implemented in Java, which ensures that the clients can run on virtually any type of machine. The server components of that application might be implemented in

C++ for high performance. CORBA makes possible the communication between these various components.

IDL, which specifies between CORBA objects, is instrumental in ensuring CORBA's language independence. Because interfaces described in IDL can be mapped to any programming language, CORBA applications and components are thus independent of the language(s) used to implement them. In other words, a client written in C++ can communicate with a server written in Java, which in turn can communicate with another server written in COBOL, and so forth.

One important thing to remember about IDL is that it is not an implementation language. That is, we can't write applications in IDL. The sole purpose of IDL is to define interfaces; providing implementations for these interfaces is performed using some other languages.

3.4 The CORBA Communication

CORBA provides a layer of abstraction above the communication layer, thus shielding clients from needing to know about the underlying communications mechanisms or where in the network the target object resides.

In order to understand CORBA communications, we must first understand its role in a network of computing systems. Typically, a computer network consists of systems that are physically connected. This physical layer provides the medium through which communication can take place, whether that medium is a telephone line, a fiber-optic cable, a satellite uplink, or any combination of networking technologies.

Somewhere above the physical layer lies the transport layer, which involves protocols responsible for moving packets of data from one point to another. So how does CORBA fit into this networking model? It turns out that the CORBA specification is neutral with respect to network protocols; the CORBA standard specifies what is known as the *General inter_ORB Protocol* (GIOP), which specifies, on a high level, a standard for communication between various CORBA ORBs and components. GIOP, as its name suggests, is only a general protocol; the CORBA standard also specifies additional protocols that specialize GIOP to use a particular transport protocol.

The *Internet Inter_ORB Protocol* (IIOP) is a specialization of the GIOP. IIOP is the standard protocol for communication between ORBs on TCP/IP based networks. An ORB must support IIOP in order to be considered CORBA 3.0 compliant.

CORBA uses the notion of object references (which in CORBA/IIOP lingo are referred to as *Interoperable Object References* or IORs) to facilitate the communication between objects. When a component of an application wants to access a CORBA object, it first obtains an IOR for that object. Using the IOR, the component (called a client of that object) can then invoke methods on the object (called the server in this instance).

In CORBA, a client is simply any application that uses the services of a CORBA object; that is, an application that invokes a method or methods on other objects. Likewise, a server is an application that creates CORBA objects and makes the services provided by those objects available to other applications.

CORBA ORBs usually communicate using the Internet Inter ORB Protocol (IIOP). Other protocols for inter-ORB communication exist, but IIOP is fast becoming the most popular, first of all because it is the standard, and second because of the popularity of TCP/IP (the network protocol using by the Internet), a layer that IIOP sits on the top of. CORBA is independent of networking protocols, however, and could (at least theoretically) run over any type of network protocol.

3.5 The CORBA Object

An object model describes how objects are represented in an object-oriented system. CORBA as an object-oriented architecture has an object model as well. But the CORBA object model probably differs somewhat from the traditional object models because of its distributed object features. Three of the major differences between the CORBA object model and traditional models lie in CORBA's semi-transparent support for object distribution, its treatment of object references, and its use of what are called object adapters [5].

In CORBA, all communication between objects is done through object references (these are known as Interoperable Object References or IORs).

In fact, it is more natural to use object-oriented technologies in the distributed computing area than it is used in non-distributed computing. This is due to the inherently decentralized nature of distributed computing.

Furthermore, visibility to objects is provided only through passing reference to those objects. In other words, remote objects in CORBA remain remote; there is currently no way for an object to move or copy itself to another location.

A CORBA object is a programming entity with an identity, an interface, and an implementation. It is capable of being located by the ORB and being invoked by clients. It does not physically exist unless made concrete by an object implementation. From a client's perspective, a CORBA Object's identity is encapsulated in the object reference. An object reference is a value that reliably denotes a particular object. It allows a client program to identity, locate and use a particular CORBA object. An object reference refers to a single CORBA Object. An object may be denoted by multiple, distinct object references.

3.5.1 Object Distribution

To a CORBA client, a remote method call looks exactly like a local method call, thanks to the use of client stubs. Thus, the distributed nature of CORBA objects is transparent to the users of those objects; the clients are unaware that they are actually dealing with objects, which are distributed on a network. Actually, the proceeding statement is almost true. Because object distribution brings with it more potential for failure (due to a network outage, server crash, and so on), CORBA must offer a contingency to handle such possibilities. It does by offering a set of system exceptions, which can be raised by any remote method.

3.5.2 Object References

In a distributed application, there are two possible methods for one application component to obtain access to an object in another process. One method is known as passing by reference. In this method, the first process, Process A, passes an object reference to the second process, Process B. When Process B invokes a method on that object, the method is executed by Process A because that process owns the object. Process B only has visibility to the object through the object reference, and thus can only request Process A to execute methods on Process B's behalf. Passing an object by reference means that a process grants visibility of one of its objects to another process while retaining ownership of that object. That is, when an object is passed by reference, the object itself remains in the place while an object reference for that object is passed. Operations on the object through the object reference are actually processed by the object itself.

The second method of passing an object between application components is known as passing by value. In this method, the actual state of the object, such as the values of its member variables, is passed to the requesting component, typically through a process known as serialization. When methods of the object are invoked by Process B, they are executed by Process B instead of Process A, where the original object resides. Furthermore because the object is passed by value, the state of the original object is not changed; only the copy (now owned by Process B) is modified. Generally, it is the responsibility of the developer to write the code that *serializes* and *deserializes* objects. Serialization refers to the encoding of an object's state into a stream, such as a disk file or network connection. When an object is serialized, it can be written to such a stream and subsequently read and deserialized, a process that converts the serialized data containing the object's state back into an instance of the object.

Currently, CORBA only supports the method of passing by reference and does not support the method of passing of objects by value.

3.6 CORBA Clients and Servers

Traditionally, in a client / server application, the server is the component, or components, which provides services to other components of the application. A client is a component that consumes services provided by a server or servers. The architecture of a CORBA application is no different; generally, contain components of an application provide services that are used by other components of the application. Not surprisingly, the general terms client and server refer to these components of a CORBA application. When considering a single remote method invocation, however, the roles of client and server can be temporarily reversed because a CORBA object can participate in multiple interactions simultaneously.

In a CORBA application, any component that provides an implementation for an object is considered a server, at least where the object concerned. If a component creates an object and provides other components with visibility to that object, that component acts as a server for that object; any requests made on that object by other components will be processed by the component that created the object. Being a CORBA server means that the component executes methods for a particular object on behalf of other components (the clients).

Frequently, an application component can provide services to other application components while accessing services from other components. In this case, the component is acting as a client of one component and as a server to the other components. In fact, two components can simultaneously act as clients and servers each other.

Like the client / server architectures, CORBA maintains the notions of clients and servers. In CORBA, a component can act as both a client and as a server. Essentially, a component is considered a server if it contains CORBA objects whose services are accessible to other CORBA objects. Likewise, a component is considered a client if it accesses services from some other CORBA object. Of course, a component can simultaneously provide and use various services, and so a component can be considered a client and/or server, depending on the scenario in a question.

3.7 Stubs and Skeletons

When implementing CORBA application components, we will encounter what are known as client *stubs* and server *skeletons.* A client stub is a small piece of code that allows a client component to access a server component. This piece of code is compiled along with the client portion of the application. Similarly, server skeletons are pieces of code that you "fill in" when you implement a server. You do not need to write the client stubs and server skeletons yourselves; these pieces of code are generated when you compile IDL interface definitions.

3.8 Basic Object Adapters (BOA)

The CORBA standard describes a number of what are called object adapters, whose primary purpose is to interface an object's implementation with its ORB. The *Basic Object Adapter* (BOA) provides CORBA objects with a common set of methods for accessing ORB functions. These functions range from user authentication to object activation to object persistence. The BOA is, in effect, the CORBA object's interface to the ORB. According to the CORBA specification, the BOA should be available in every ORB implementation, and this seems to the case with most CORBA products available. One particularly important feature of the BOA is its object activation and deactivation capability. The BOA supports four types of activation policies, which indicate how application components are to be initialized. These activation policies include the following:

- The shared server policy: in which a single server is shared between multiple objects.

- The unshared server policy: in which a server contains only one object.
- The server per method policy: which automatically starts a server when an object method is invoked and exits the server when the method returns.
- The persistent server policy: in which the server is started manually.

The variety of activation policies allows an application architect to choose the type of behavior that makes the most sense for a particular type of server. For instance, a server requiring a length of time to initialize itself might work best as a persistent server, because the necessary initialization time would aversely affect the response time for that server. On the other hand, a server that starts up quickly upon demand might work well with the server per method policy.

3.9 Portable Object Adapter (POA)

The BOA provides a bare minimum of functionality to server applications. As a consequence, many ORBs added custom extensions to the BOA to support more complex demands upon an object adapter, making server implementations incompatible among different ORB vendors. Portable Object Adapter (POA) provides a much-extended interface that addresses many needs that were wished for, but not available with the original BOA specification. The POA features include:

- Support for transparent activation of objects.
- Servers can export object references for not-yet-active servants that will be incarnated on demand.
- Allow a single servant to support many object identities.
- Allow many POAs in a single server, each governed by its own set of policies.

- Delegate requests for non-existent servants either to a default servant, or ask a servant manager for an appropriate servant.

The general idea is to have each server contain a hierarchy of POAs. Only the root POA is created by default; a reference to the root POA is obtained using the resolve_initial_references() operation on the ORB. New POAs can be created as the child of an existing POA, each with its own set of policies.

3.10 CORBA Services and Facilities

ORB enables you to write application interfaces to represent those components that implement specific tasks for an application. Three points must be noted:

- One component may support many interfaces.
- One application is typically composed of many components.
- New application components can be built by modifying existing components.

Recall that ORB allows components to communicate with each other. Because everything runs and depends on ORB, ORB is essential to creating distributed applications. Unfortunately, ORB alone is not enough to create distributed applications: you typically need services to locate components and manage their life cycles; you may need system management services to observe the health of the system; and you may even need domain specific interfaces and frameworks to help you do rapid development. OMG thus provides additional capabilities in the form of services and facilities that provide both horizontal and vertical capabilities. Horizontal services generally are useful to all industries, whereas vertical services are designed to meet specific industries' needs.

3.10.1 CORBA Services

Object services are domain-independent horizontal interfaces that are used in most distributed applications. They are available in the form of CORBA interfaces and come with implementations. Without object services, writing distributed applications would not be easy. Object services are collectively called *CORBAservices.*

CORBA provides 15 types of object services, each of which is described briefly in the following list:

- *Naming service*: Provides a client with the capability to obtain an IOR of another component anywhere on the bus. Also allows a component to bind its name to a naming context, an object that contains a set of name bindings in which each name is unique.
- *Event service*: Supports asynchronous even notifications and event delivery. Allows component to register dynamically their interest in an event. The design of this service is scalable and is suitable for distributed environments.
- *Life cycle service*: defines operations to create, copy, move, and remove components on ORB.
- *Persistent object service* (POS): Provides a set of generic interfaces for storing and managing the persistent state of components. The component ultimately has the responsibility of managing its state, but it can use or delegate to POS for the actual work. A major characteristic of POS is its openness – it allows a variety of different clients and implementations of POS to work together.

- *Transaction service*: Supports a two-phase commit protocol between components. Supports two transaction models – the flat and nested models.
- *Concurrency control service*: Enables multiple clients to coordinate their access to shared resources. This service is useful for keeping a resource in a consistent state when multiple, concurrent clients access it.
- *Relationship service*: Allows components to form dynamic relationships (links) between each other. This service defines two new kinds of objects: relationships and roles. A role represents a CORBA component in a relationship. One potential use of this service is to create workflow managers.
- *Externalization service*: Defines protocols for externalizing and internalizing component data. This service enables a component to externalize its state in a stream of data (in memory, on a disk file, or across the network) and then internalize into a new instance of the component in the same or a different server process.
- *Licensing service*: Enables component vendors to control the use of their intellectual property. This service itself does not define any business policy. Vendors can define this service according to their own requirements and the requirements of their customers.
- *Query service*: Enables a component to invoke queries on collections of other components. The queries can be used to invoke arbitrary operations.
- *Property service*: Provides a mechanism to associate properties (named values) dynamically with components. This service defines operations to create and manipulate sets of name-value pairs.

- *Security service*: Ensures secure communication between components by providing authentication, authorization, auditing, non-repudiation, and administration (for example, a security policy).
- *Time service*: Enables the user to obtain current time in a distributed environment. The service can be used to determine the order in which events occurred, and compute the interval between two events.
- *Collections service*: Provides a uniform way to create and manipulate the most common collections of components.
- *Trader services*: Enables a client to obtain an IOR of another component on ORB based on some component properties. (Note that "properties" has nothing to do with the property service.)

3.10.2 CORBA Facilities

Like object services, common facilities are interfaces. Printing facilities, database facilities, system management facilities, and email facilities are some examples of common facilities that are horizontally oriented.

OMG also provides domain interfaces, which are vertically oriented interfaces for application domains, such as finance, health care, manufacturing, telecom, electronic commerce, and transportation. Common facilities are collectively called *CORBAfacilities*.

Chapter 4 Building CORBA Applications

4.1 Benefits of using CORBA

There are three benefits to be obtained by building applications using CORBA:

- Coding is quicker, so applications can be deployed sooner: Since the software developer doesn't have to write as much code, development happens faster.
- Applications designed around discrete services have better architecture: Good architecture divides applications into modules or object groups based on functionality. By designing around the architecture, which is itself based on this principle, your application gets a head start on its own sound architecture.
- Many CORBA implementations have enterprise characteristics built in: they're robust, and they scale: Providers know where you're going to deploy your applications with their CORBA products, so they compete to meet your enterprise's needs. Scalable name servers, transaction services, and other services have proved their worth in enterprise deployments for years already!

4.2 Typical steps to build a CORBA application

The outline of building a CORBA application process is as follows:

- Define the server's interfaces using IDL
- Choose an implementation approach for the server's interfaces. Normally, CORBA provides two such approaches: inheritance and delegation.

- Use the IDL compiler to generate client stubs and server skeletons for the server interfaces.
- Implement the server interfaces.
- Compile the server application.
- Run the server application.
- Build a client that uses the services you implemented.

4.2.1 Building a CORBA Server

The first step in building a CORBA application is usually to implement the server functionality. The reason for this is that while a server can be tested, at least in limited way, without a client, it is generally much more difficult to test a client without a working server. There are exceptions to this, of course, but typically you will need to at least define the server interfaces before implementing the client; server and client functionality can then be developed in parallel.

4.2.2 Building a CORBA client

Conceptually speaking, the process of building the client is much simpler; you only need to decide what you want the client to do, include the appropriate client stubs for the types of server objects you want to use, and implement the client functionality. Then you will be ready to compile and run the client.

4.2.3 Choosing an Implementation Approach

Before actually implementing the server functionality, you will first need to decide on an implementation approach to use. CORBA supports two mechanisms for implementation of IDL interfaces. Developers familiar with object-oriented concepts might recognize these mechanisms, or at least their names. These include the inheritance mechanism and the delegation mechanism.

Implementation by inheritance consists of a base class that defines the interfaces of a particular object and a separate class, inheriting from this base class, which provides the actual implementations of these interfaces. That is, the implementation class derives from a class provided by the IDL compiler.

Implementation by delegation consists of a class that defines the interfaces for an object and then delegates their implementations to another class or classes.

The primary difference between the inheritance and delegation approaches is that in delegation, the implementation classes need not derive from any class in particular.

Chapter 5 VisiBroker

VisiBroker is a complete CORBA2.3 Object Request Broker (ORB) that supports the development, deployment, and management of distributed object applications across a variety of hardware platforms and operating systems [2].

The client and server programs deployed with Visibroker ORBs are shown in Figure 3.

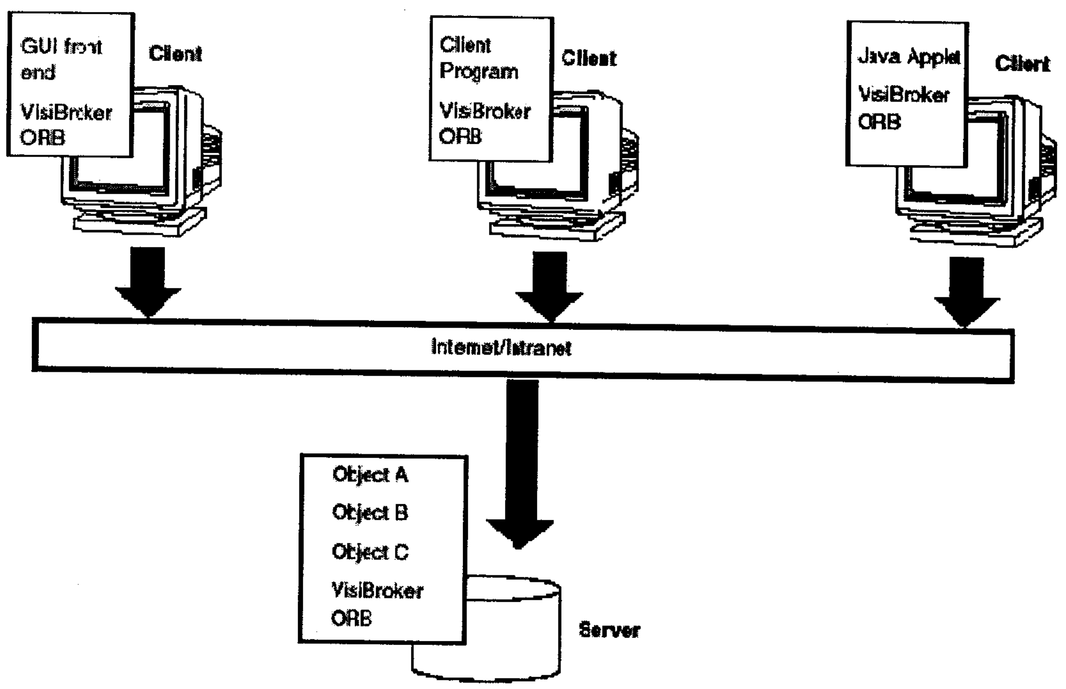


Figure 3

5.1 Product

In addition to VisiBroker (the ORB), three other components are available with this product. They include

- Naming Service

  The Naming Service allows you to associate one or more logical names with an object implementation and store those names in a namespace. It also lets client applications use this service to obtain an object reference using the logical name assigned to that object.

- Event Service

  The Event Service provides a facility that decouples the communication between objects. It provides a supplier-consumer communications model that allows multiple supplier objects to send data asynchronously to multiple consumer objects through an event channel.

- Gatekeeper

  Gatekeeper runs on a web server and enables client programs to locate and use objects that do not reside on the web server and to receive callbacks, even when firewalls are being used. The Gatekeeper can also be used as an HTTP daemon, thereby eliminating the requirement for a separate HTTP server during the application development phase.

5.2 VisiBroker Runtime Package

The VisiBroker Runtime Package, in conjunction with a Java or C++ runtime environment, enables client and server applications to use and offer distributed objects.

The Runtime Package is a subset of the Development Environment and is required to deploy an application [1]. Components of the Runtime Package include the following:

- ORB: The runtime library needed by servers and clients.
- Smart Agent: The *Smart Agent* (osagent) is used by applications to locate the objects they wish to use. It is a process that must be started on at least one host within the network.
- Location Service: The Location Service allows VisiBroker applications to locate all instances of an object programmatically. Working with the Smart Agents on your network, the Location Service can help clients with load balancing by providing information on all the available instances of an object to which a client can bind.
- Object Activation Daemon: The *Object Activation Daemon* (OAD) enables objects to be activated automatically when they are needed by a client application. This reduces overhead by allowing servers that implement objects for client applications to be started on demand, rather than running continuously.
- Interface Repository: The *Interface Repository* is an online database of meta-information about object types. Meta information stored for ORB objects includes information about modules, interfaces, operations, attributes and exceptions. Applications that use the dynamic interfaces require that an Interface Repository be available.

5.3 Developing applications with VisiBroker

When we develop distributed applications with VisiBroker, we must first identify the objects required by the application. We will then usually follow these steps:

- Write a specification for each object using the Interface Definition Language (IDL).
- Use the IDL compiler to generate the client stub code and server POA servant code. Using the idl2cpp compiler, we'll produce client-side stubs and server-side classes (which provides classes for the implementation of the remote objects).
- Write the client program code.
- Write server object code.
- Compile the client and server code.
- Start the server.
- Run the client program.

These steps are shown in Figure 4.

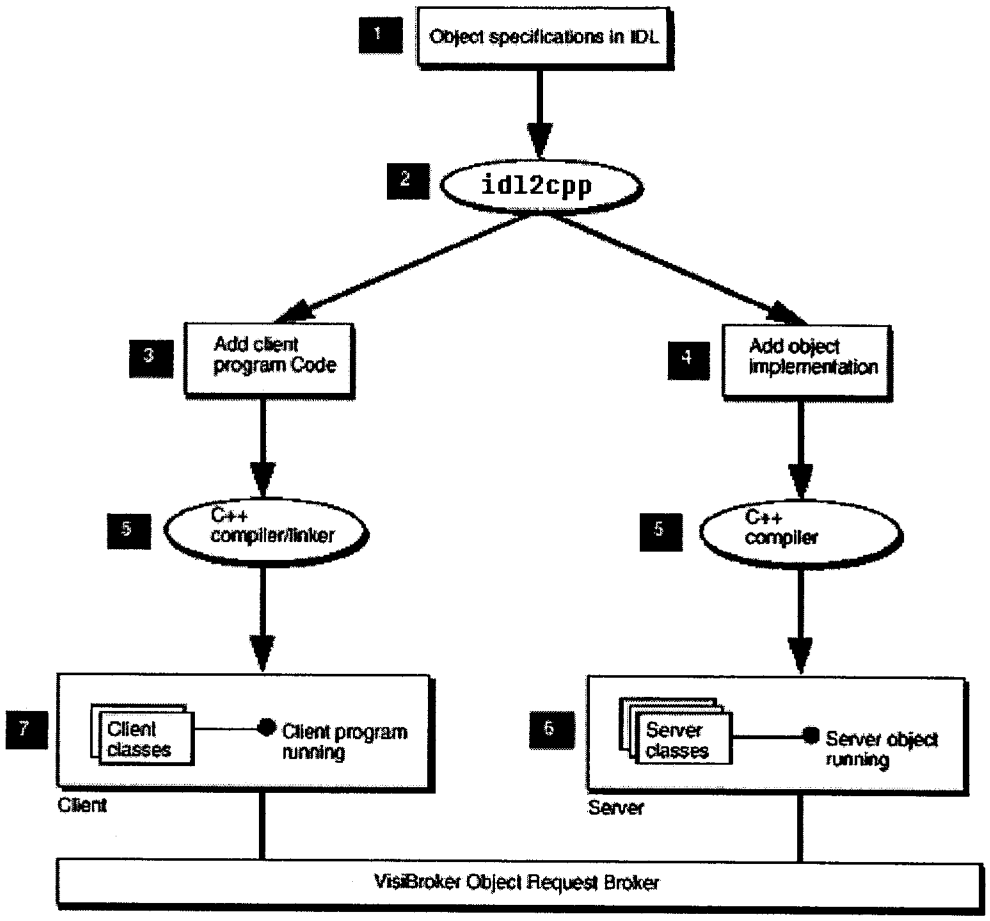

1
Object specifications in IDL
2
idl2cpp
3
Add client program Code
4
Add object implementation
5
C++ compiler/linker
5
C++ compiler
7
Client classes
Client program running
Client
6
Server classes
Server object running
Server
VisiBroker Object Request Broker

Figure 4

Chapter 6 An application developed with the VisiBroker

We would like to use the VisiBroker to build a game application as an example to describe why we design the game application using the CORBA architecture, how to develop a CORBA application and some implementation issues for a CORBA application.

6.1 The Gomoku game

The game is described as following:

- There is a board like an international chess board. It has vertical and horizontal lines on it, dividing the board into N * N squares. These squares are represented by an N * N matrix in the program. The appearance of the board is shown in Figure 5.

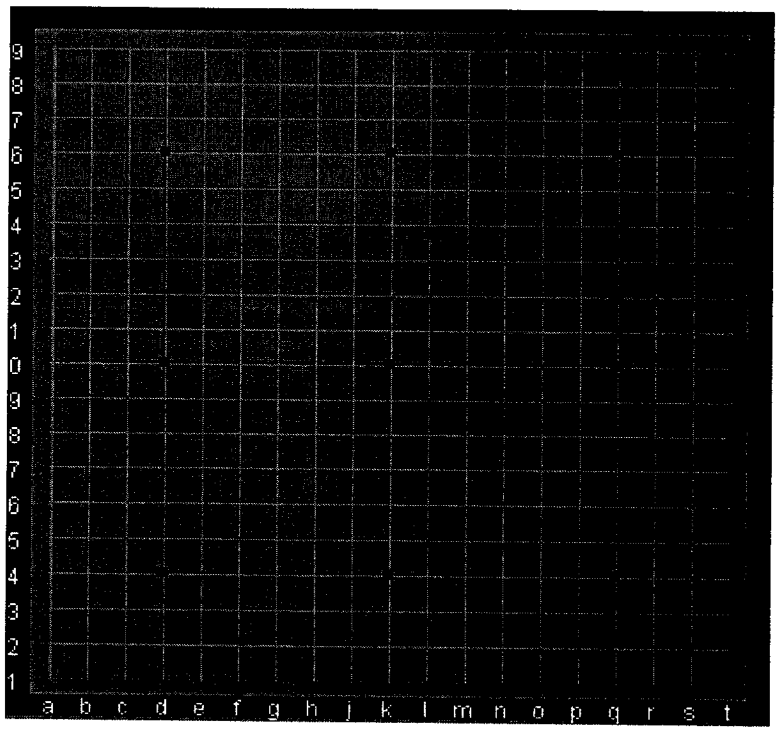


Figure 5

- There are two kinds of pieces (beans). One is white color, and the other one is black color.
- There are two opponent players, each with his own color pieces.
- The opponents move alternately.
- One movement is just put one of his own color pieces in an available position on the board. An available position means any intersection point of vertical and horizontal lines but there is no piece in that position up to now.
- The winner is the person who uses his pieces to make up a line of five continuous pieces first in a line that may be any vertical, horizontal or diagonal line.

6.2 The design methodology

A good systems analysis begins by capturing the requirements of an application, and modeling the essential elements in its environment. The design follows the features below:

- The game application is completely object-oriented design.
- Object-oriented language C++ is used.
- Distributed computing environment is perfect for a game application.

6.2.1 Advantages of using object-oriented design methodology

Object-oriented technology has gained rapid acceptance among software developers and has become the preferred choice for designing and implementing software systems because object orientation has proven to be an adequate means for developing and maintaining large scale applications. It seems reasonable to apply the object paradigm to distributed computation as well. Object-oriented technology supplies abstraction, encapsulation, inheritance and polymorphism. By using *Object-Oriented Design* (OOD) technology to develop software, we have the following advantages:

- Natural: OOD is a natural way of thinking about problems and finding solutions to the problems.
- Simple: OOD helps to decompose a complex application system into smaller, simpler pieces, enabling a problem to be divided and conquered.
- Reusable: Inheritance and composition enable programmers to plug in the existing objects into the new code that is being developed.

- Replaceable: Existing objects can be readily replaced with new implementations, as long as the new object supports the same interfaces as the object that it replaces.
- Extensible: New capabilities can be added to existing objects easily.
- Strong cohesion: A well-designed object, by definition, is cohesive because its data and behavior are always related.
- Loose coupling: The interfaces between objects define object' contract that enables loose coupling and independent development.

6.2.2 Advantages of using C++ as the implementation language

C++ is an object-oriented language. All the expressions of object orientation are naturally captured in the C++ language. For example, C++ class characterizes abstraction and encapsulation, C++ class derivation implements inheritance, and C++ virtual mechanism dynamic binding achieves polymorphism.

C++ users can rely on CORBA compliant ORB to help them develop portable distributed OO applications using C++ in a natural fashion.

6.2.3 Advantages of using distributed object computing for a game application

- The distributed computing environment is required by our game application in which it operates.
- The game clients and game servers run on separate computers.

- The computers can be linked through heterogeneous inter networks of LANs and WANs.
- Clients and servers may be heterogeneous end system platforms.

6.3 Sequence diagram

Sequence diagrams describe interactions among classes in terms of an exchange of messages within the collaboration to effect a desired operation or result over time.

A sequence diagram has two dimensions; the vertical dimension represents time, the horizontal dimension represents different objects. Normally time proceeds down the page.

We use the sequence diagram to detail the requirements of the game application and to describe the interactions among the classes.

The Sequence Diagram with concurrent objects of the game is as shown in Figure 6.

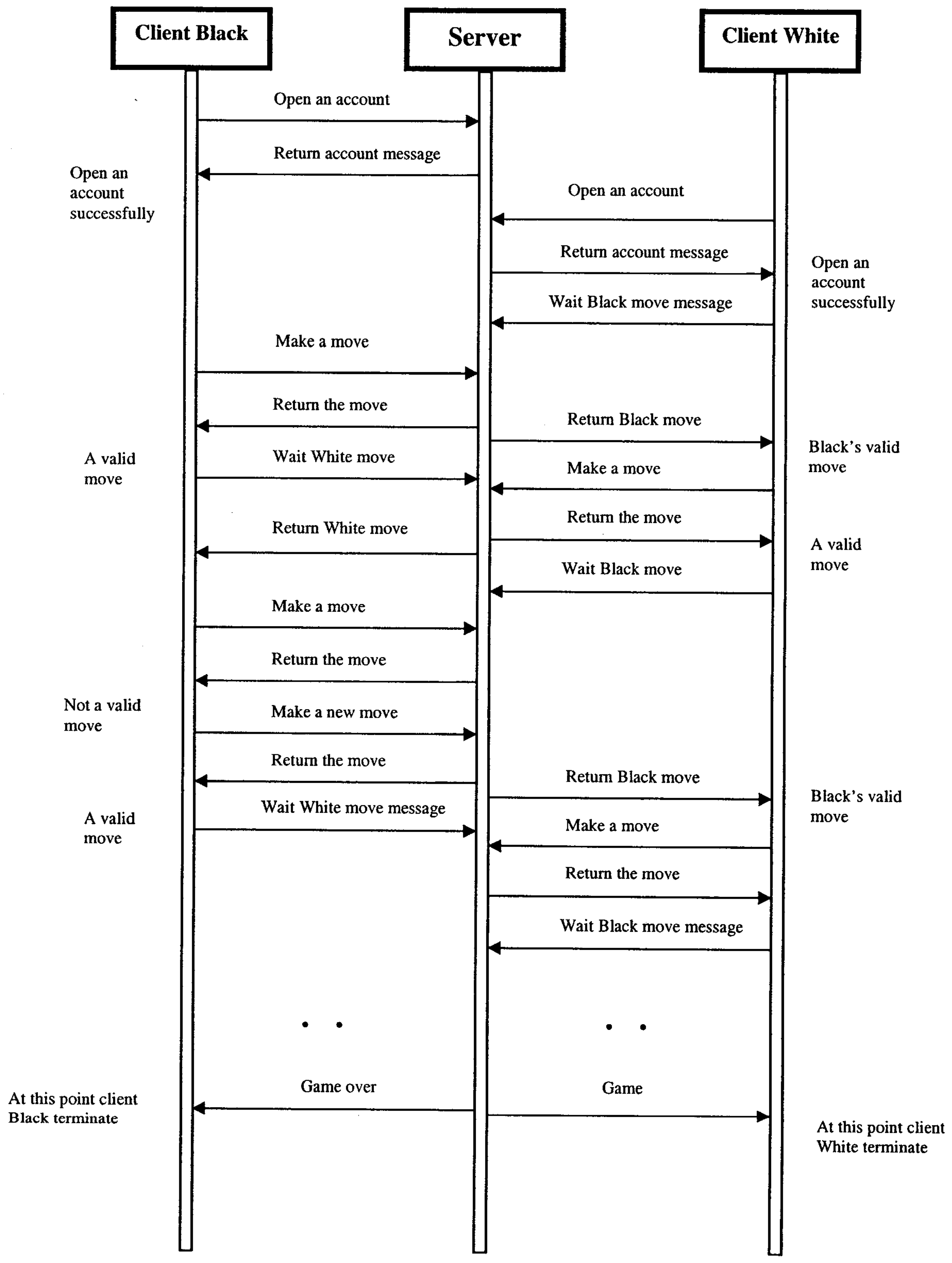

Client Black
Server
Client White
Open an account
Return account message
Open an account successfully
Open an account
Return account message
Open an account successfully
Wait Black move message
Make a move
Return the move
Return Black move
A valid move
Wait White move
Black's valid move
Make a move
Return White move
Return the move
A valid move
Wait Black move
Make a move
Return the move
Not a valid move
Make a new move
Return the move
Return Black move
Black's valid move
A valid move
Wait White move message
Make a move
Return the move
Wait Black move message
Game over
Game
At this point client Black terminate
At this point client White terminate

Figure 6

6.4 The architecture of the application

As above sections described, this game application is most suitable to use the CORBA architecture and to be designed as a CORBA application.

- There is a server and two or more clients. Two of the clients are players, and the other clients can observe their playing.
- The clients can be on different machines and different platforms as long as the machines are on the network and a CORBA product is installed on the machines. Of course, the client program of the game application must be installed on the machines.
- Clients must register on the server. Any move of a player on his local machine will be sent to the server, that is, clients do not communicate with each other directly, and they communicate through the server.
- The clients make requests to the server through ORB and networks just like the server running on the same machine.

Object-oriented design technology is also used for the game design because of the features of object-oriented design technology as above sections described and also because:

- CORBA itself and applications built on top of it are designed using object-oriented software development principles. The management of complexity afforded by OO software development techniques is very important for the practical implementation and deployment of CORBA applications.

- It is more natural to use object-oriented technologies in the distributed computing area than it is used in non-distributed computing. This is due to the inherently decentralized nature of distributed computing.

The logical structure of the architecture is as shown in Figure 7.

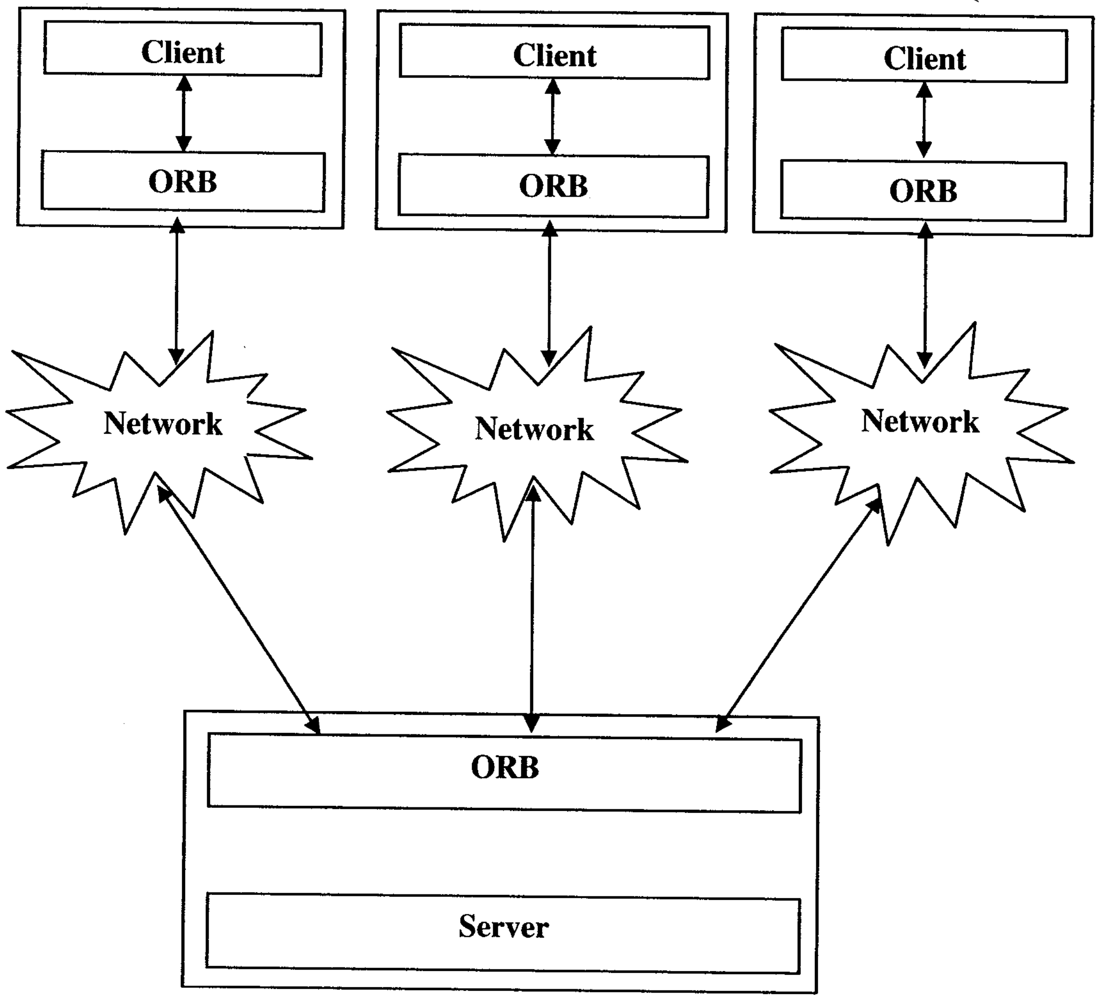


Figure 7

6.5 The design of the application

We will use the class diagrams of UML (Unified Modeling Language) to describe the design. Class diagrams are the backbone of almost every object-oriented method, including UML. They describe the static structure of a system.

For the IDL interfaces, there is one module called Game, which contain two interfaces. The class diagram for the interfaces is as shown in Figure 8:

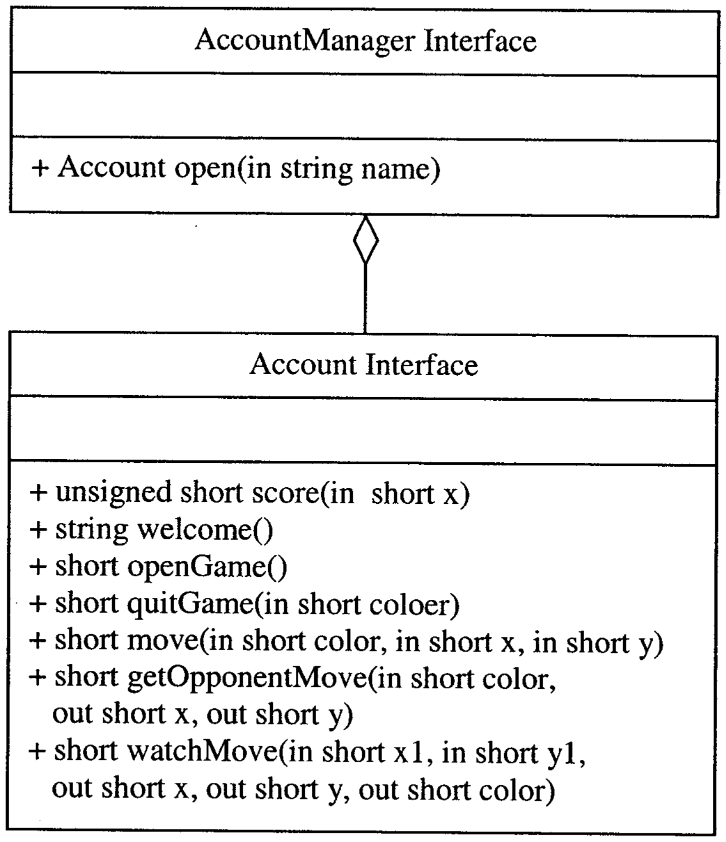

AccountManager Interface
+ Account open(in string name)
Account Interface
+ unsigned short score(in short x)
+ string welcome()
+ short openGame()
+ short quitGame(in short coloer)
+ short move(in short color, in short x, in short y)
+ short getOpponentMove(in short color,
out short x, out short y)
+ short watchMove(in short x1, in short y1,
out short x, out short y, out short color)

Figure 8

The class diagram for the client side is as shown in Figure 9:

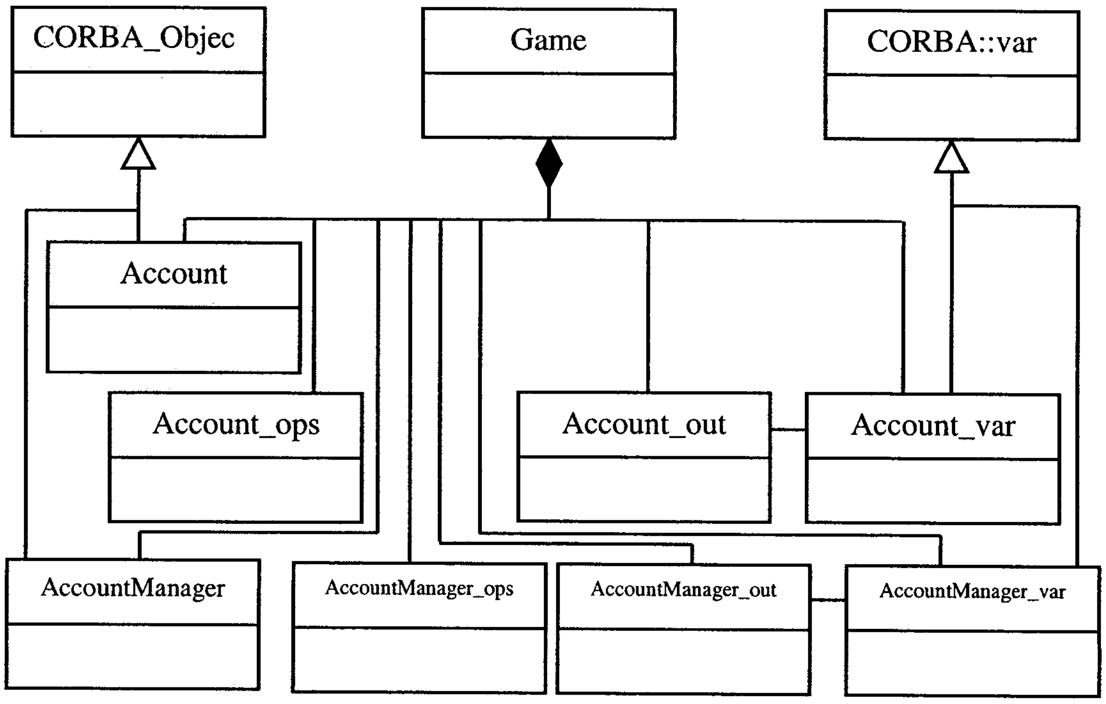


Figure 9

The class diagram for the server side is as shown in Figure 10:

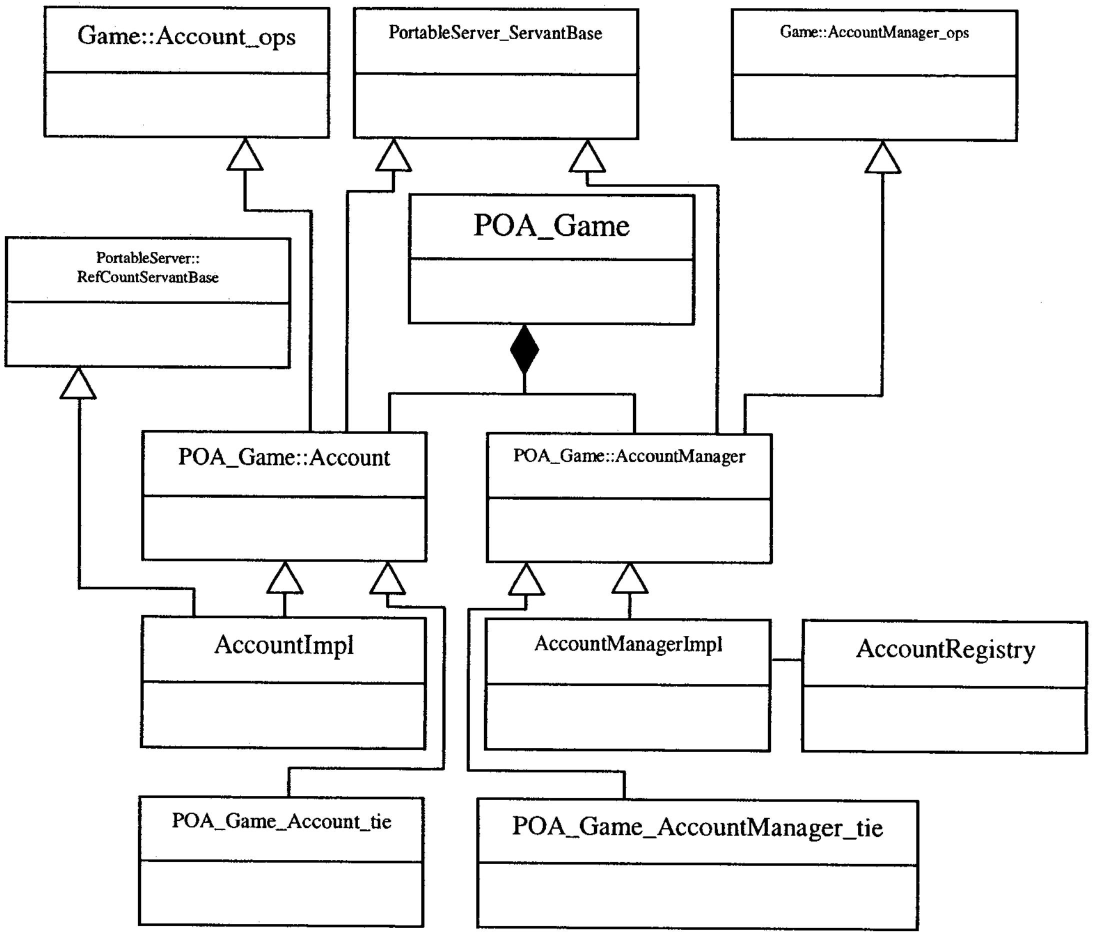


Figure 10

6.6 The detail design and implementations

The VisiBroker 4.0 for C++ language is used for implementing both client side and server side under Windows NT, but both the client side and the server side programs can be run under any platform when the CORBA product is installed in that platform because only ANSI Standard C++ is used for the implementations.

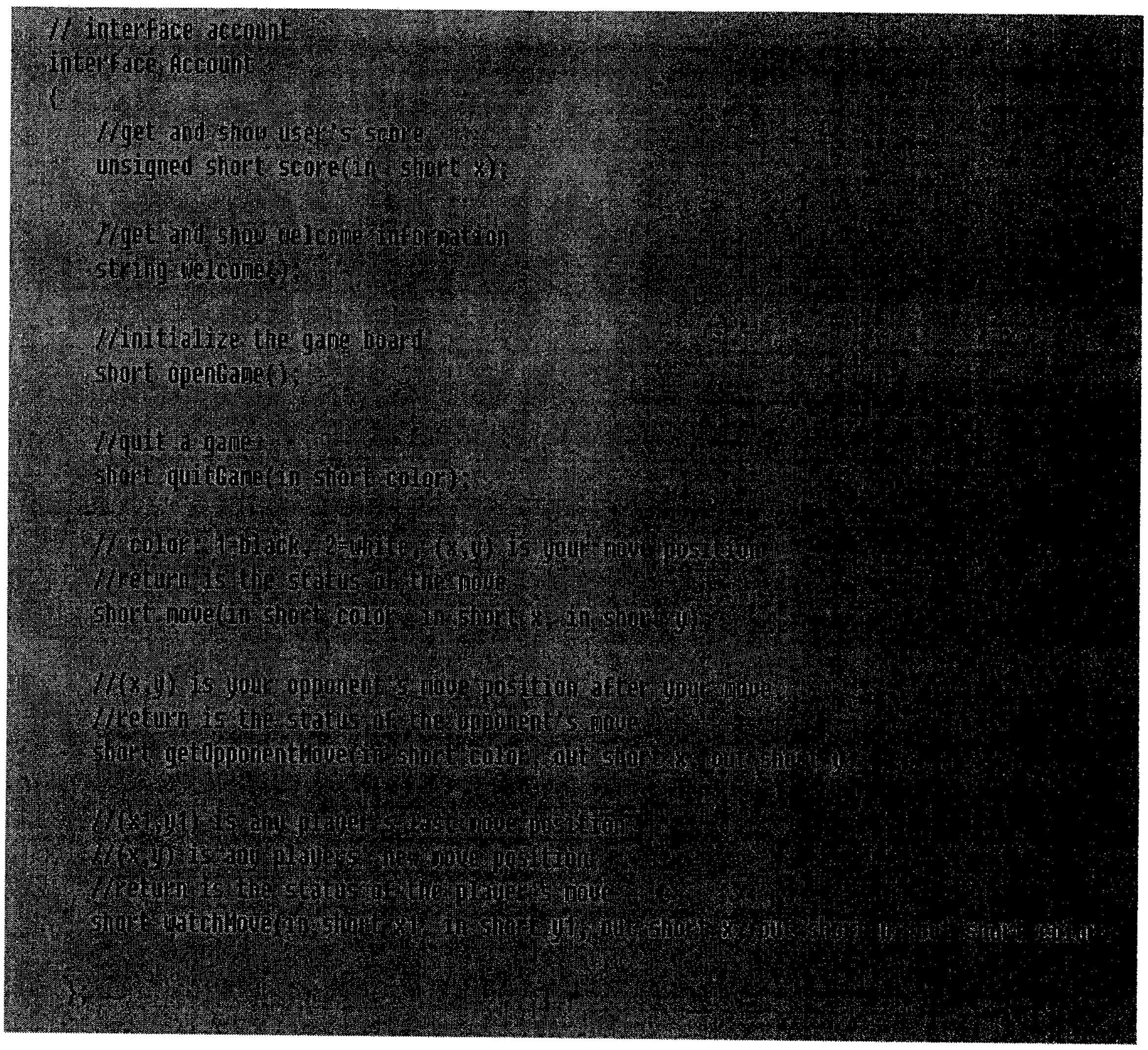
6.6.1 MakeFiles

Two makefiles MakeFile.cpp and StdmMk_NT are used to make whole compiles and links easer from IDL, client and server C++ source codes to executable codes for both client and server sides.

6.6.2 Define interfaces

There are two interfaces that are defined in a module with IDL language for the game in the file Game.idl. One is called Account, and the other is called AccountManager. Each interface contains some methods that clients can call through the object reference. Interface Account mainly contains the methods to play the game, and interface AccountManager mainly contains the methods for registration for a player in order to play a game. The syntax of IDL language is like C++/Java syntax.

An interface defined grammatically for the game is shown in Figure 11.



```
// interface account
interface Account
{
    //get and show user's score
    unsigned short score(in short x);

    //get and show welcome information
    string welcome();

    //initialize the game board
    short openGame();

    //quit a game
    short quitGame(in short color);

    // color: 1=black, 2=white; (x,y) is your move position
    //return is the status of the move
    short move(in short color, in short x, in short y);

    //(x,y) is your opponent's move position after your move
    //return is the status of the opponent's move
    short getOpponentMove(in short color, out short x, out short y);

    //(x1,y1) is any player's last move position
    //(x,y) is any player's new move position
    //return is the status of the player's move
    short watchMove(in short x1, in short y1, out short x, out short y, out short color);

};
```

Figure 11

The IDL compiler "idl2cpp" of VisiBroker for C++ takes IDL file (which contains Account interface) as input and produces the necessary client stubs (which provide the interface to the Account objects' methods used by client program for all member function invocations on the client side) and server skeletons (which provide classes for the implementations of the remote objects on the server side) in C++ [3].

The "idl2cpp" compiler generates four files from the Game.idl file:

- Game_c.hh: Contains the definitions for the Account and AccountManager classes used to build the client program.
- Game_c.cpp: Contains internal stub routines used by the client.
- Game_s.hh: Contains the definitions for the AccountPOA and AccountManagerPOA servant classes used to build server objects.
- Game_s.cpp: Contains the internal routines used by server.

6.6.3 Implementing the server

The Server.c file is a server implementation. This file implements the server classes for the server side of the game. The server program does the following:

1. Initializes the Object Request Broker.

   The ORB provides a communication link between the client and the server. When a client makes a request, the ORB locates the object implementation, delivers the request to the object, and returns the response to the client. Each application must initialize the ORB before communicating with it.

2. Creates and sets up a Portable Object Adapter with the required policies.

   The Portable Object Adapter (POA) and its components determine which servant should be invoked when a client request is received, and invokes that servant. A servant is a programming object that provides the implementation of an abstract object. A servant is not a CORBA object.

   One POA called the root POA is supplied by each ORB. We can create additional POAs and configure them with different behaviors and can also define the characteristics of the objects the POA objects –the POA controls.

The steps to setting up a POA with a servant include:

- Obtaining a reference to the root POA

  All server applications must obtain a reference to the root POA to manage objects or to create new POAs. We can obtain a reference to the root POA by using function resove_initial_references(). The resove_initial_references() returns a value type CORBA::Object. We can then use this reference to create other POAs if they are needed.

- Defining the POA policies

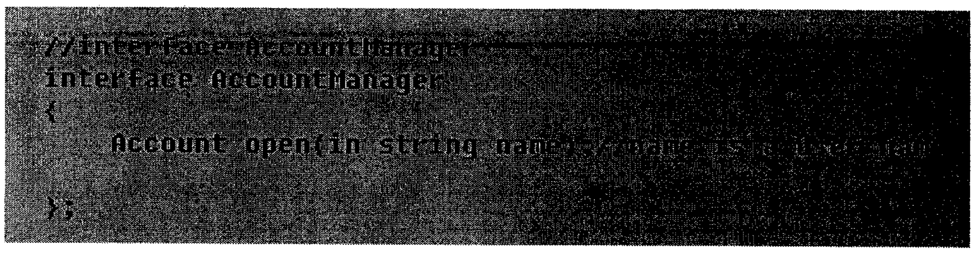
  The root POA has a predefined set of policies that cannot be changed. A policy is an object that controls the behavior of a POA and the objects the POA manages. If we need a different behavior, such as different lifespan policy, we will need to create a new POA.

- Creating a POA as a child of the root POA

  POAs are created as children of existing POAs using create_POA(). We can create as many POAs as are required. Children POAs do not inherit the policies of their parent POAs.

3. Creates the account manager servant object.

   The servant object must be created and activated in the server side.When we compile an IDL that contains an interface, a class is generated that serves as the base class for the servant. For example, in Game.idl file, an AccountManager interface is defined as shown in Figure 12.



```
//interface AccountManager
interface AccountManager
{
    Account open(in string name); //name is a user [illegible]

};
```

Figure 12

A base class POA_Game::AccountManager is generated. Then the implementation class AccountManagerImpl is derived from the base class.

4. Activates the servant object.

   There are several ways in which objects can be activated:

   - Explicit—all objects are activated upon server start up via calls to the POA.
   - On demand—the servant manager activates an object when it receives a request for a servant not yet associated with an object ID.
   - Implicit—objects are implicitly activated by the server in response to an operation by the POA, not by any client request.
   - Default servant—the POA uses the default servant to process the client request.

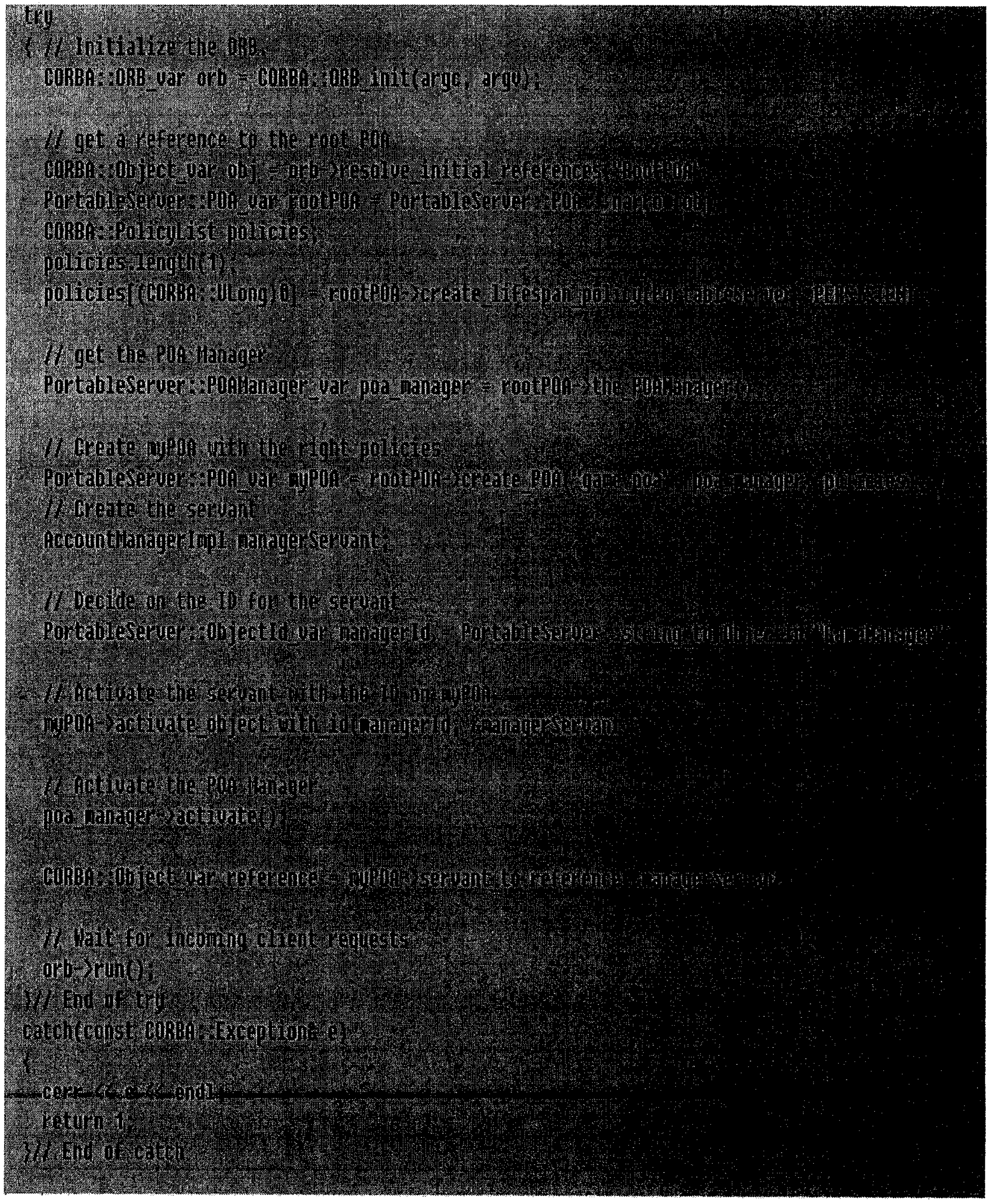
5. Activate the POA manager and the POA.

   A POA manager is associated with a POA during POA creation. By default, POA Managers are created in a holding state. In this state, all requests are routed to a holding queue and are not processed. To allow requests to be dispatched, the POA manage associated with the POA must be changed from the holding state to an active state. A POA manager is simply an object that controls the state of the POA.

6. Waits for incoming client requests.

Once the POA is set up, the server can wait for client requests by using function orb.run(). This process will run until the server is terminated.

The main steps of the server program for the game are as shown in Figure 13.



```
try
{ // Initialize the ORB.
  CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

  // get a reference to the root POA
  CORBA::Object_var obj = orb->resolve_initial_references("RootPOA[illegible]
  PortableServer::POA_var rootPOA = PortableServer::POA::_narrow(obj[illegible]
  CORBA::PolicyList policies;
  policies.length(1);
  policies[(CORBA::ULong)0] = rootPOA->create_lifespan_policy(PortableServer::PERSISTENT[illegible]

  // get the POA Manager
  PortableServer::POAManager_var poa_manager = rootPOA->the_POAManager([illegible]

  // Create myPOA with the right policies
  PortableServer::POA_var myPOA = rootPOA->create_POA("game_poa", poa_manager, policies[illegible]
  // Create the servant
  AccountManagerImpl managerServant;

  // Decide on the ID for the servant
  PortableServer::ObjectId_var managerId = PortableServer::string_to_ObjectId("GameManager"[illegible]

  // Activate the servant with the ID on myPOA
  myPOA->activate_object_with_id(managerId, &managerServant[illegible]

  // Activate the POA Manager
  poa_manager->activate();

  CORBA::Object_var reference = myPOA->servant_to_reference(&managerServant[illegible]

  // Wait for incoming client requests
  orb->run();
}// End of try
catch(const CORBA::Exception& e)
{
  cerr << e << endl;
  return 1;
}// End of catch
```

Figure 13

The account class is derived from the POA_Game::Account class that is generated by the idl2cpp compiler from the Account interface. Look closely at the POA_Game::Account class definition that defines in the Game_s.hh file and notice that it is derived from the Account class. The class hierarchy is as shown in Figure 14.

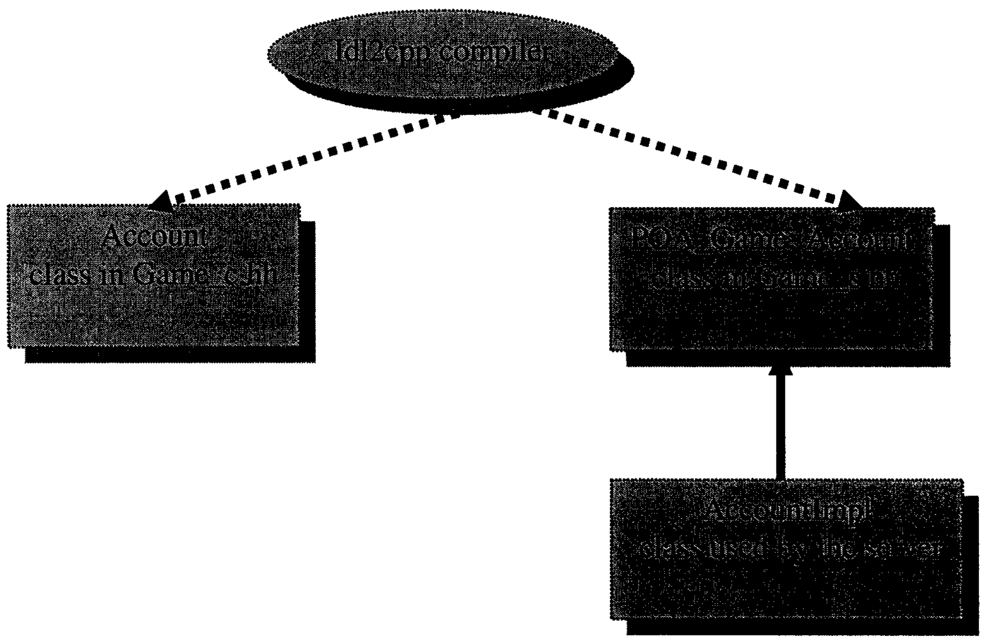
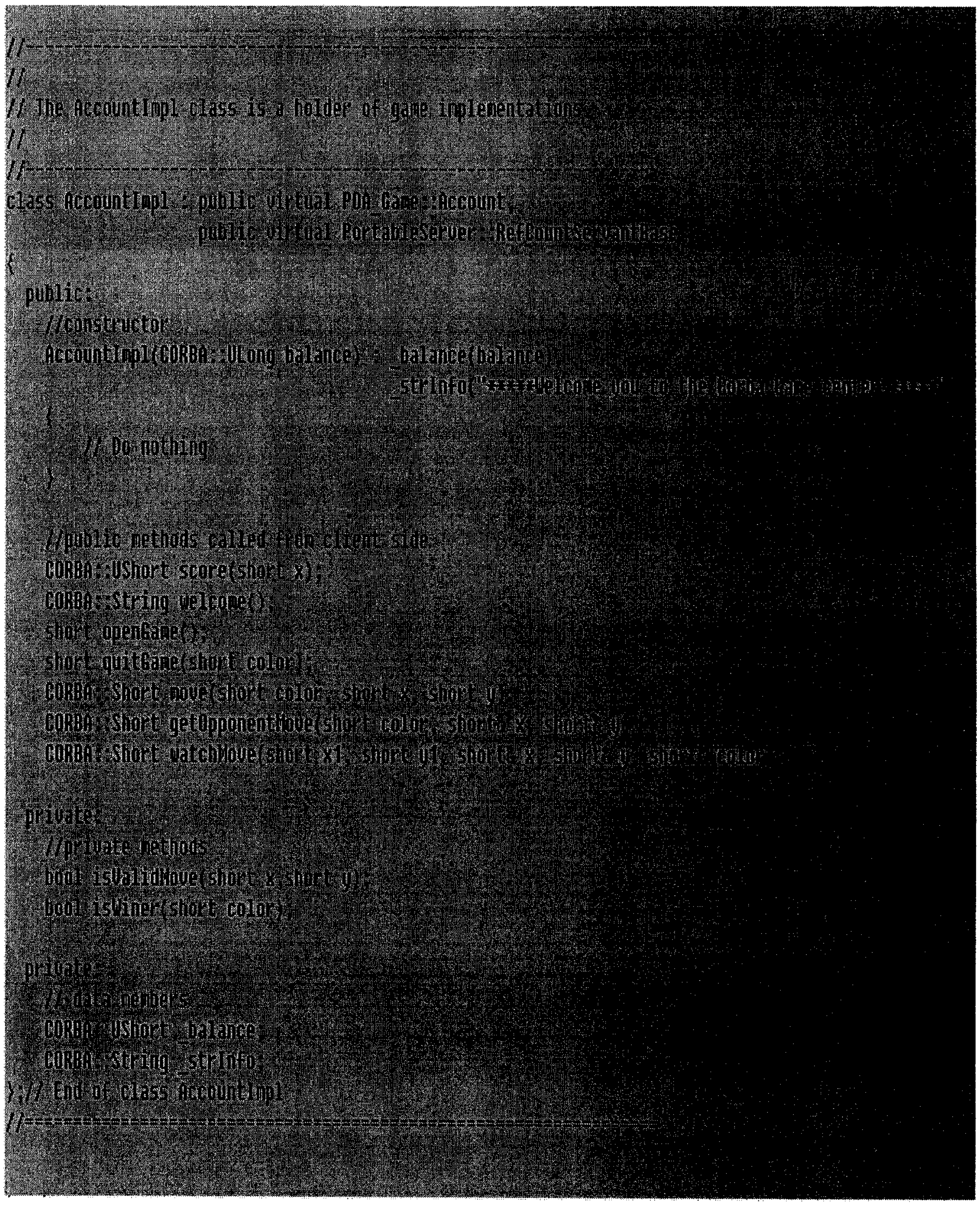


Figure 14

The AccountImpl class is a holder of game implementations. It inherits POA_Game::Account and PortableServer::RefCountServantBase and implements most of game operations (methods) except registration operations. (The registration operations are defined in another class called class AccountRegistry.) These methods are called from client side through the object references and ORB.

The AccountImpl class is defined as shown in Figure 15.

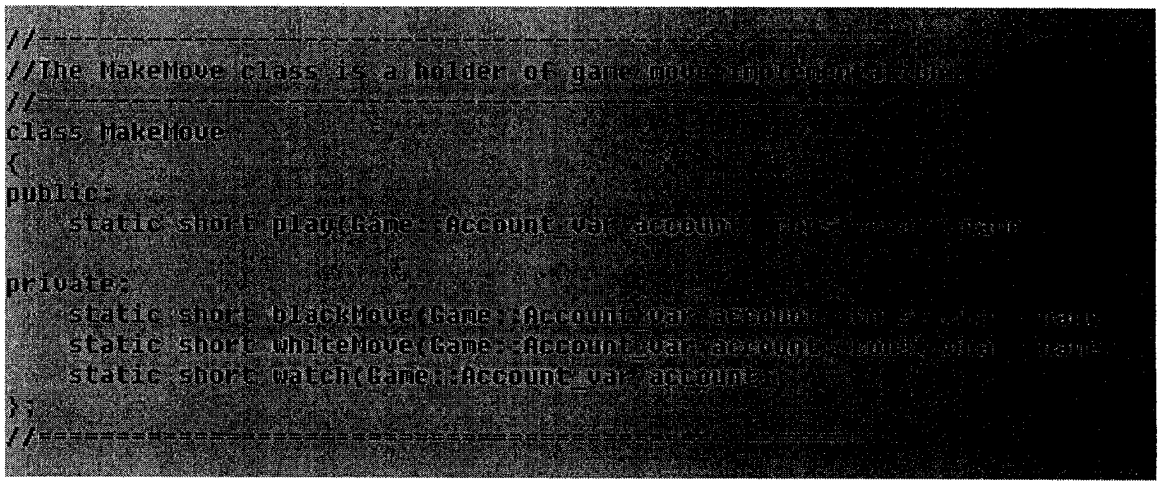


```
//------------------------------------------------------------
//
// The AccountImpl class is a holder of game implementations
//
//------------------------------------------------------------
class AccountImpl : public virtual PDA_Game::Account,
                    public virtual PortableServer::RefCountServantBase
{
 public:
  //constructor
  AccountImpl(CORBA::ULong balance) : _balance(balance),
                                      _strInfo("*****Welcome you to the [illegible]
  {
      // Do nothing
  }

  //public methods called from client side
  CORBA::UShort score(short x);
  CORBA::String welcome();
  short openGame();
  short quitGame(short color);
  CORBA::Short move(short color, short x, short y);
  CORBA::Short getOpponentMove(short color, short& x, short& y);
  CORBA::Short watchMove(short x1, short y1, short& x, short& y, short& color

 private:
  //private methods
  bool isValidMove(short x,short y);
  bool isWiner(short color);

 private:
  // data members
  CORBA::UShort _balance;
  CORBA::String _strInfo;
};// End of class AccountImpl
//============================================================
```

Figure 15

6.6.4 Implementing the client

The client side program is in the client.cpp file that contains game move implementations as shown in Figure 16.



```
//------------------------------------------------------------
//The MakeMove class is a holder of game move [illegible]
//------------------------------------------------------------
class MakeMove
{
public:
    static short play(Game::Account_var account [illegible]

private:
    static short blackMove(Game::Account_var account [illegible]
    static short whiteMove(Game::Account_var account [illegible]
    static short watch(Game::Account_var account [illegible]
};
//============================================================
```

Figure 16

Many of the classes used in implementing the game client are also contained in the Game_c.hh and Game_c.cpp files.

The client program uses an object reference to invoke an operation on the object that has been defined in the object's IDL interface and has been implemented on the server side object. But before player can make a move, the game client program performs the following steps:

- Initializes the ORB

  Not only the server needs to initialize the ORB, but also the client program needs to explicitly initialize the ORB. The ORB is transparent to the client. That is the client is unaware that the object may be on the same machine or across a network. Function CORBA::ORB_init() is used to initialize the ORB.

- Binding to Objects

  A client program uses a remote object by obtaining a reference to the object. Object references are usually obtained by using the <interface>_bind() member function. The ORB hides most of the details involved with obtaining the object reference, such as locating the server that implements the object and establishing a connection to that server.

  A client uses the method Game::AccountManager::_bind() to bind to the AccountManager object, which gets the manager ID, locates an account manager, gives the full POA name and the servant ID.

  When the server process starts, it performs a CORBA::ORB.init() and announces itself to smart Agents on the network.

  When the client program invokes _bind() member function, the ORB performs server functions on behalf of the client program.

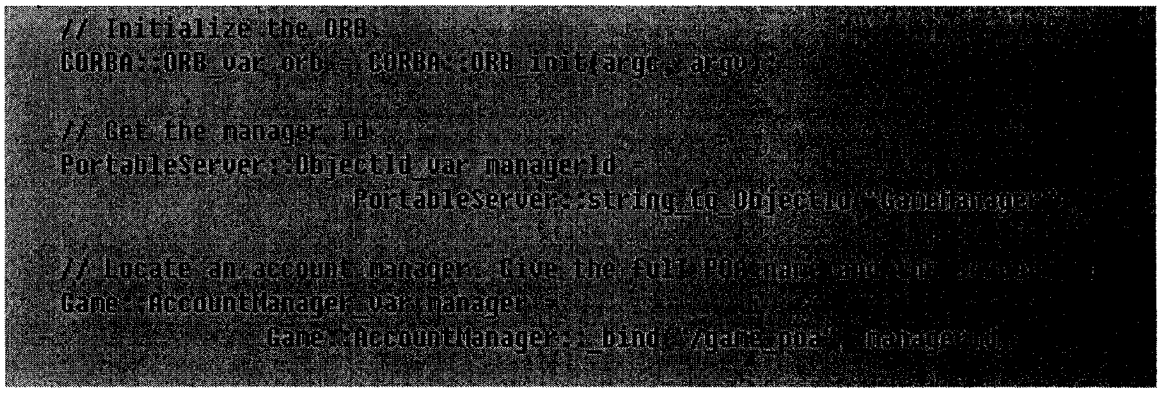
  - ORB contacts the Smart Agent to locate an object implementation that offers the requested interface.
  - When an object implementation is located, the ORB attempts to establish a connection between the object implementation that was located and the client program.
  - Once the connection is successfully established, the ORB will create a proxy object and return a reference to that object. Then, the client will

invoke methods on the proxy object that will, in turn, interact with the server object.

  - o The client program will never invoke a constructor of the server class. Instead, an object reference is obtained by invoking the static _bind() member function.

- Requests the account manager to open an account for the named player by invoking the open() member function. Then, the player can start to play as a black piece holding player, a white piece holding player or as a watcher to watch a game played by other players.

The way for initializing ORB and binding to objects of the game client program is as shown in Figure 17.



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

// Get the manager Id
PortableServer::ObjectId_var managerId =
                PortableServer::string_to_ObjectId("GameManager");

// Locate an account manager. Give the full POA name and [illegible]
Game::AccountManager_var manager =
            Game::AccountManager::_bind("/game_poa", managerId);
```

Figure 17

6.7 The application running environment

The game program for both server side and client side can be run on any platform for which a CORBA product (for instance, VisiBroker, Orbix or OmniOrb etc.) is installed.

6.7.1 The Smart Agent

Before we attempt to run VisiBroker client programs or server implementations we must first start the Smart Agent on at least one host in the local network.

VisiBroker's Smart Agent is a dynamic, distributed directory service that provides facilities used by both client programs and object implementations. The Smart Agent locates the specified implementation so that a connection can be established between the client and the implementation. The communication with the Smart Agent is completely transparent to the client programs and also completely transparent to the object implementations. VisiBroker locates a Smart Agent from a client program or object implementation by using a broadcast message. The first Smart Agent who responds to the broadcast will be used. After a Smart Agent has been located, a point-to-point UDP connection is used for sending requests to the Smart Agent.

Chapter 7 Conclusion

An important characteristic of large computer networks such as the Internet and corporate intranets is that they are heterogeneous. CORBA has proven itself as a solid basis for heterogeneous object-oriented distributed systems.

This report has described the Common Object Request Broker Architecture portion of the OMG Object Management Architecture (OMA). CORBA provides a flexible communication and activation substrate for distributed heterogeneous object-oriented computing environments. The strengths of CORBA include:

- Heterogeneity: The use of OMG IDL to define object interfaces allows these interfaces to be used from a variety of programming languages and computing platforms.
- Object Model: The Object Model and Reference Model provided by the OMA define the rules for interaction between CORBA objects such that the interactions are independent of underlying network protocols. CORBA-based applications are abstracted away from the networking details and thus can be used in a variety of environments.
- Legacy integration: The CORBA specification is flexible enough to allow ORBs to incorporate and integrate existing protocols and applications such as DCE or Microsoft COM, rather than replace them.
- Object-oriented approach: CORBA itself and applications built on top of it are designed using object-oriented software development principles. The management

of complexity afforded by OO software development techniques is very important for the practical implementation and deployment of CORBA applications.

It is clear that the future of CORBA is very promising. The flexibility and adaptability offered by CORBA make it very attractive for distributed applications. CORBA has also an advantage over other distributed object computing technologies (such as DCOM and Java RMI) since it can be integrated into a wider range of platforms.

This report also discusses other distributed computing approaches, and briefly compares these popular distributed object paradigms. The architectures of CORBA, DCOM and Java/RMI provide mechanisms for transparent invocation and accessing of remote distributed objects. Though the mechanisms that they employ to achieve remote distribution may be different, the approach each of them takes is more or less similar.

The CORBA-based application of *gomoku* game is given as an example to show how the CORBA architecture is used to develop distributed object-based applications.

This example also shows how CORBA helps to reduce the complexity of developing distributed applications.

We like to note that our game approach has no restrictions for only this *gomoku* game. The approach can be used to implement most games and similar distributed object applications.

References

[1] VisiBroker Version 4.0 Installation Guide
(Inprise) http://www.borland.com/visibroker/

[2] VisiBroker Version 4.0 for C++ Programmer's Guide
(Inprise) http://www.borland.com/visibroker/

[3] VisiBroker Version 4.0 for C++ reference
(Inprise) http://www.borland.com/visibroker/

[4] Aklecha, Vishwajit, Object-Oriented Frameworks Using C++ and CORBA,
Coriolis technology press, 1999

[5] Rosenberger, Jeremy L., Teach Yourself CORBA in 14 Days
Sams, 1998

[6] Object Management Group, The Common Object request Broker: Architecture and specification 2.0, 2.3, 3.0.
http://www.omg.org/

[7] Henning, Michi and Vinoski, Steve, Advanced CORBA Programming with C++, Addison-Wesley, 1999

[8] Ahmed, Suhail, CORBA Programming Unleashed, SAMS, 1998

[9] Orfali, Robert and Harkey, Dan, Client/Server Programming with Java and CORBA Wiley, 1998

[10] Schettino, John, O'Hara, Liz, and Hohman, Robin S. CORBA for Dummies, IDG Books, 1998

[11] www.corba.org/

[12] www.cs.wustl.edu/~schmidt/corba.html

[13] www.mico.org/

[14] http://www.theaceorb.com/

[15] http://www.uk.research.att.com/omniORB/omniORB.html

[16] http://www.omg.org/gettingstarted/

[17] http://cbbrowne.com/info/corba.html

[18] http://my.execpc.com/gopalan/misc/compare.html

[19] http://www.iona.com/products/orbix.htm

[20] http://www.ois.com/resources/corb-9.asp

[21] http://www.sei.cmu.edu/str/descriptions/corba.html

[22] http://research.microsoft.com/~ymwang/papers/HTML/DCOMnCORBA/S.html

[23] http://www.scit.wlv.ac.uk/~cm1924/cp3025/distrib/reading/corba/corba3/corba6.htm

Appendix

How to run the application

The game program for both server side and client side can be run on any platform for which a CORBA product (for instance, VisiBroker, Orbix or OmniOrb etc.) is installed.

To run the game program in the VisiBroker environment, the following three steps must be executed:

1. Starting the Smart Agent
2. Starting the Server
3. Starting the clients

1 Starting the Smart Agent

Before we attempt to run VisiBroker client programs or server implementations we must first start the Smart Agent on at least one host in the local network.

The basic command for starting the Smart Agent is as follows.

Under Windows NT platform, open a DOS Command Prompt window, and then start a Smart Agent by using the following DOS command:

Prompt> osagent

Under UNIX platform, start a Smart Agent by using the following command:

Prompt> osagent &

If we are running Windows NT and want to start the Smart Agent as an NT Service, we need to register the ORB Services as a NT Service during installation. If the Service is registered, we then are able to start the Smart Agent as an NT Service through the Service Control Panel.

2 Starting the server

Under Windows NT platform, open a DOS Command Prompt window, and then start the game server by using the following DOS command:

Prompt>start server

Under UNIX platform, start the game server by using the following command:

Prompt>Server&

3 Starting the clients

Under Windows NT platform, open a separate DOS Command window, which can be on different computers, for each client and then start a client program by using following DOS Commands:

- To start a client program as a player holding black pieces:

  Prompt> client black

- To start a client program as a player holding white pieces:

  Prompt> client white

- To start a client program as a watcher to watch other player's play a game. We start as many clients as we want for watching a game:

  Prompt> client <Name>

Under UNIX platform, start a client program is the same way as start a client under Windows NT.

List of Acronyms

ANSI:	American National Standard Institute
API:	Application Programming Interface
BOA:	Basic Object Adapter
CORBA:	Common Object Request Broker Architecture
DCE:	Distributed Computing Environment
DCOM:	Distributed Component Object Model
DII:	Dynamic Invocation Interface
GIOP:	General inter_ORB Protocol
IDL:	Interface Definition Language
IIOP:	Internet Inter_ORB protocol
IOR:	Interoperable Object Reference
JRMP:	Java Remote Method Protocol
JVM:	Java Virtual Machine
OMA:	Object Management Architecture
OMG:	Object Management Group
OOD:	Object-Oriented Design
OOP:	Object-Oriented Programming
ORB:	Object Request Broker
OSF:	Open Software Foundation
POA:	Portable Object Adapter

RPC:	Remote Procedure Call
RMI:	Remote Method Invocation
TCP/IP:	Transfer Control Protocol/Internet Protocol
UDP:	User Datagram Protocol
UML:	Unified Modeling Language
URL:	Uniform Resource Locator