THE CHINESE UNIVERSITY OF HONG KONG
Department of Computer Science and Engineering

Part-time Master Programme in Computer Science
CSC 7251/7260 Project

Web-based Learning with CORBA

By
POON Ping-yeung

under
the supervison
of
Professor Michael Lyu

Abstract

Common Object Request Broker Architecture (CORBA) is the most important and ambitious middleware technologies ever undertaken by the software industry. CORBA uses objects as a unifying metaphor for bring existing applications to the bus. At the same time, it provides a solid foundation for a component-based future.

Web based learning is trend in learning technology. Under the current atmosphere of developing both the Internet infrastructure and content for wider use in Hong Kong, it would be an important topic to study.

We will first discuss the importance of web based learning in the current trend in learning system. We will then introduce the CORBA, Object Management Group (OMG) and Object Management Architecture (OMA). The advantages of using Java in CORBA architecture is explained. One of the CORBA product – Visibroker for Java being the product to be used in the implementation will be introduced. The specification and the implementation will be explained. Two server deployment features were further discussed. Finally, we will come up with a conclusion discussing the upsides and downsides of CORBA and particularly in web based learning environment.

Content

Introduction

Network computing become important when computers connected to each other. More and more computers connect to the Internet to form the largest network in the world. Applications worked together in the network or Internet to form distributed system. On the other hand, computer application goes for the way of object-oriented design for easier design and maintenance. When they merge together, we have “Distributed Object Computing”.

World Wide Web becomes the choice of information delivery in recent year. Java is an object-oriented language. Java can build portable object-oriented applications to run in multiple platforms. The ability to download Java applets and the close integration of Java with web browsers make it an ideal medium for web and Internet based development. CORBA is a set of specification for technologies to support distributed object computing. CORBA specify the interfaces to allow remote objects to interact. When CORBA, Java and web meet each other, they form a good team. The architectural roles they play in building distributed object systems are naturally complementary.

Being a part-time tutor in the Open University of Hong Kong, I am interested in distance learning. The web is a good media for the learning and teaching. Web-based learning will prevail in the next millennium. In this project, I will try to build a system for the management of web-based learning using CORBA, Java and web.

Web-based Learning

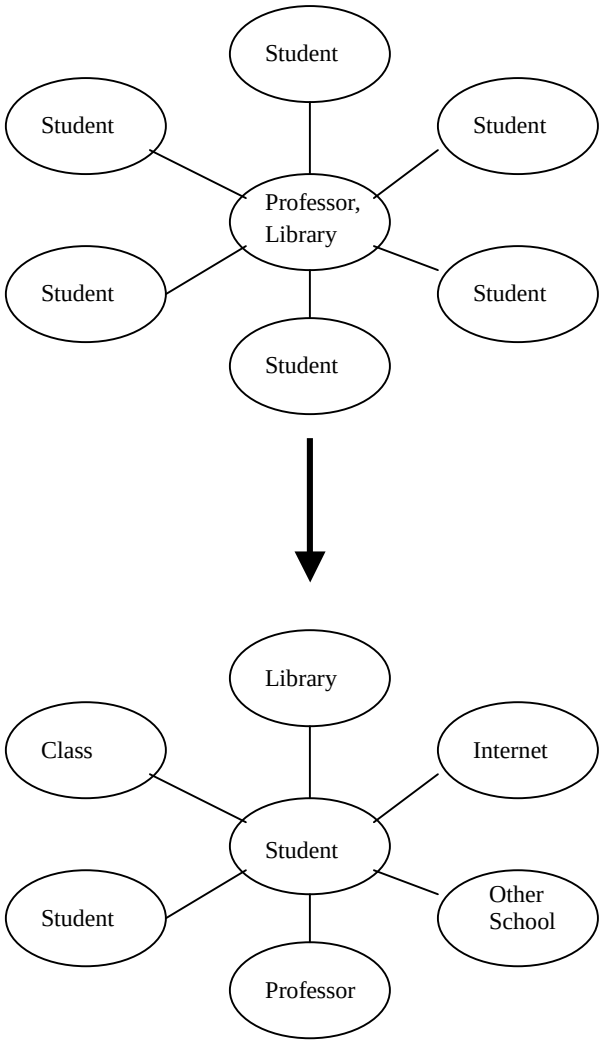
Introduction

Education and training need is increasing. Educated and skilled people are essential to economic competitiveness. The period during which certain skills are useful is shortening, so the need for lifelong learning is growing.

To enhance the effectiveness of education and training programs, people have pointed to the value of information technology. Students and working people alike perform better when using information technology. Students become more engaged in the learning process, and communication increases between students and teachers and among students. Information technology can provide new and powerful learning experiences, tailored to the needs of the learner, that go far beyond traditional classroom/teacher models. In addition, information technology potentially can reduce the time and cost required to produce and distribute educational content—educational materials can be more accessible to more people. Information technologies that promote learning can be a wise investment—and a promising means of attaining long-sought achievement levels.

Learner-centered Environment

Learners need to develop the capacity to search, select and synthesize cast amounts of information to create knowledge. More and more universities are moving from an institutionally-centered model towards a learner-centered (student-centered) approach.



In order to produce a more efficient, effective and learner-centered environment, there are several factors to be taken into account which includes cognition, collaboration, and communication. We learn from cognitive science about the way of student's learning and the obstacles they face during learning. Learning experience is defined as the interactions which includes interaction with information, interaction with instructor and interaction with other students. Besides, communication is a essential component in interaction. With the advance in technology, development of new instructional model is

facilitated the availability of a reliable network infrastructure and computer access. This made us possible to move from the classroom model instruction into a distributed learning environment.

Web-based Environment

Although much multimedia educational software is commercially available, they are rather static and only provide one to one communication. They are difficult to scale up to use in large and diverse learning environment. The software is usually proprietary and the course content is non-reusable for other course. Search tools and knowledge management are insufficient.

The advent of the World Wide Web was a breakthrough in terms of defining a standard for delivery of information independent of software and hardware system. The World Wide Web has the potential to revolutionize instruction and increase educational opportunities. A number of benefits were identified:

- ✓ Faster Training
- ✓ Reduce/ no travel costs
- ✓ Training cost per head is reduced
- ✓ Just-in-time learning
- ✓ Learning where you work
- ✓ Extended access through Internet
- ✓ Interactive and customizable content
- ✓ Active learning and student-centered

Technology

Although both the public and private sectors can benefit from the enhanced learning performance and a large amount of technology is available to support learning. The commercial solution for learning in distributed environment is still rare to find. There are many reasons to this situation. The instructional systems are still costly and complex to produce. The educational software and systems are still not easily available for many learner and educators to use. That causes obstacles to educational institutions. Educational systems are increasingly interactive and difficult to manage at the institutional level. Essential business models and key transactions in educational institution are not yet adapted to computers and distributed systems. Educational networks are still not reliable enough to become alternative solutions to traditional classroom teaching or standalone applications.

This project will focus on the possible solution to deal with some of these issues by building a prototype of web-based learning system as example.

Common Object Request Broker Architecture (CORBA) Overview

What is CORBA?

Common Object Request Broker Architecture (CORBA) is an industry standard by the Object Management Group (OMG) for creating distributed object systems. This standard allows application to communicate with each other irrespective of their running platforms and the programming language used to implement them.

What is the OMG?

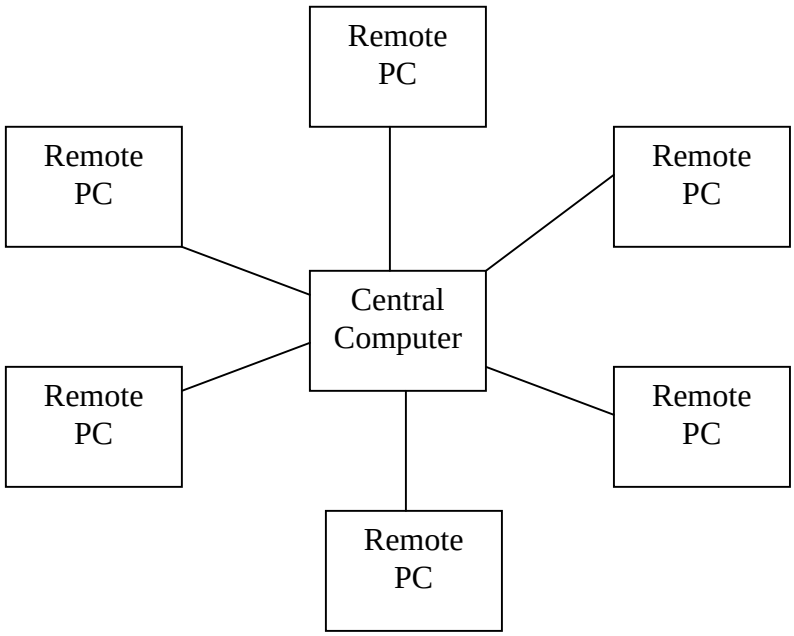
The Object Management Group (OMG) was a consortium of object technology vendors and users, whose mission is to define the architecture of an open software bus on which object components can inter-operate across networks and operating systems. It was originally founded by eight companies: 3Com Corporation, American Airlines, Canon, Inc., Data General, Hewlett-Packard, Philips Telecommunications N.V., Sun Microsystems and Unisys Corporation. In October 1989, OMG began independent operations as a non-profit corporation. Through the OMG's commitment to developing technically excellent, commercially viable and vendor independent specifications for the software industry, the consortium now includes over 800 members. The OMG moves forward in establishing CORBA as the "Middleware that's Everywhere" through its worldwide standard specifications.

What is Distributed Objects System?

A distributed object is an object that located somewhere in the network and it can be accessed as if it is a local object. The distributed objects may be located in the local or remote machine running in heterogeneous platforms. Systems that feature such objects are termed distributed objects systems. In client/server world, the client may access a number of distributed objects without knowing their actual location. An example can be like this. You are running a personal finance management system and you need to evaluate your investment portfolio. You need the current stock price to compute the value of your portfolio. The application just sends request to the stock object to obtain the latest price. The application does not need to know exactly where the price come from. The stock object may change its data sources any time. Similarly, you can calculate your tax for last year by invoking a tax object. Your application can invoke an operation in the tax object to calculate the tax to pay. The user or client applications do not need to know the details of the tax calculation and where the object is located. This also show that the design allow clear separation of business logic and the client application. The business logic that is implemented in the distributed objects can be used in many different applications.

When we compare the distributed object system with centralized system, the benefits are much more obvious. A number of drawbacks of the centralized system are:

- ✧ When the central computer fail, the whole system will be unavailable.
- ✧ The bandwidth requirements will be larger as all the data need to send to central computer for processing even if the data is only need locally.
- ✧ A larger powerful central computer is more costly to build than a number of small computer with same total processing power.



CORBA Basics

CORBA Architectures

At the heart of the CORBA is the Object Request Broker (ORB). It is the middleware that allow the communication between the client and object.

A client can use the ORB to invoke a CORBA object which resides anywhere in the network. When the client invoke a operation on a object, the ORB will intercept the call and locate the object which implement the request and pass the parameters to the object and then return the results to client. The client invokes an operation on remote objects as if it is invoking an operation on a local object. The client does not need to know where the remote objects are located. It is the location transparency which is at the center of CORBA.

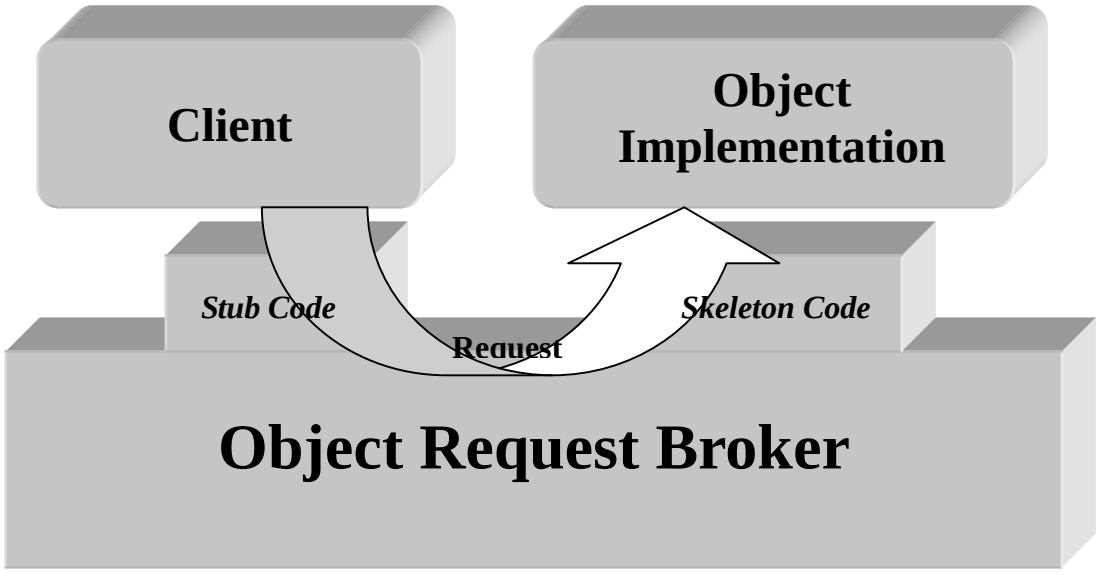
Different objects may be located on different machine and running on different operating system. The ORB can allow the objects to be implemented in different programming languages and it is up to the developer to choose the most appropriate programming language to use. That may be depend on the skills of programmers, the task to be implemented or other third party support to the language.

The purpose of all such things is to enable the interoperability between applications on different machines in a heterogeneous distributed environment.

In traditional client/server application, developers will use different kind of protocol for the communication between the client and server. The protocol used will be depend on the programming language and also the network in used. With ORB, the protocol is defined through the application interfaces via a single implementation language-neutral interface definition language, OMG IDL.

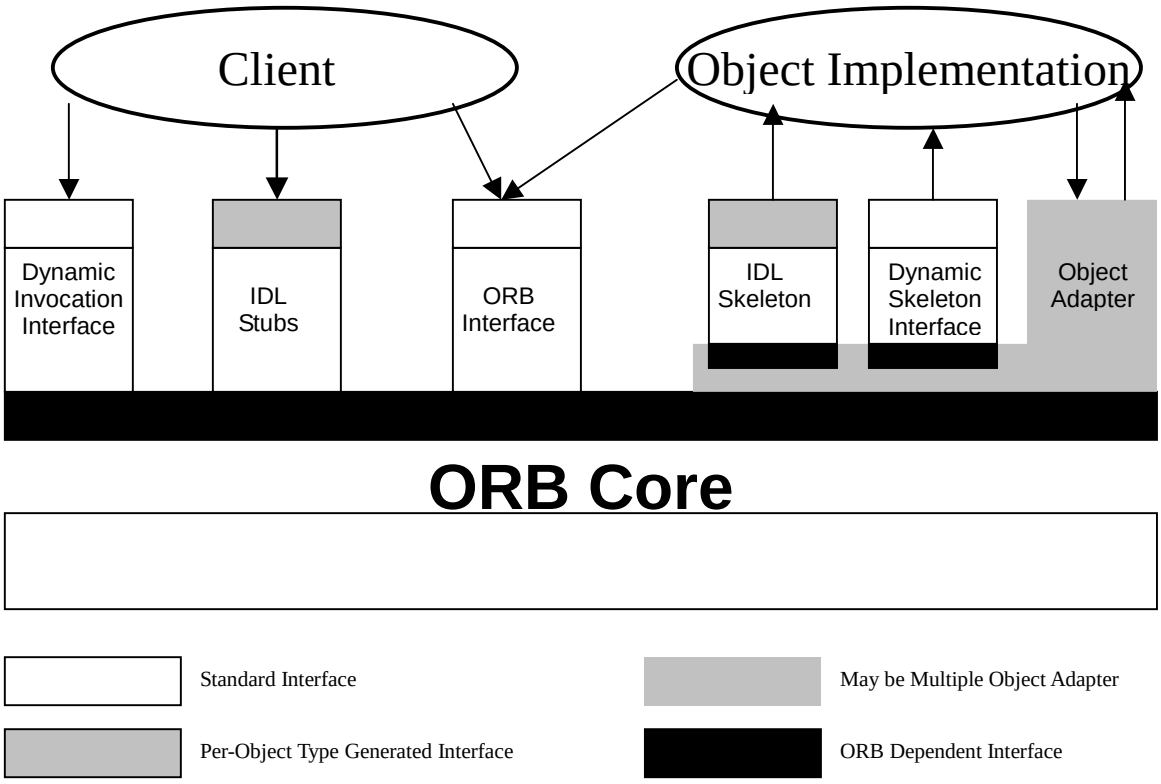
The IDL interface defined the interface of the objects including the operation the object supported, the type of parameters and the return types. The client programs only need to be written to invoke the operation of the remote objects. A IDL compiler will provide a corresponding language mapping from the IDL to the target programming language. In this way, the client can use the data type as defined in the IDL.

The IDL compiler will produce stub code and skeleton code. The stub code will link to the client and it marshal the data types of the programming language into a type suitable for transmission to the object implementation. The skeleton code will unmarshal the request from the client into suitable programming language data types for use by the object implementation. Similarly, the result from the object can also be passed back to the client in the reverse way. The following diagram illustrate a simplified view of the invocation using ORB, stub and skeleton:



The ORB Structure

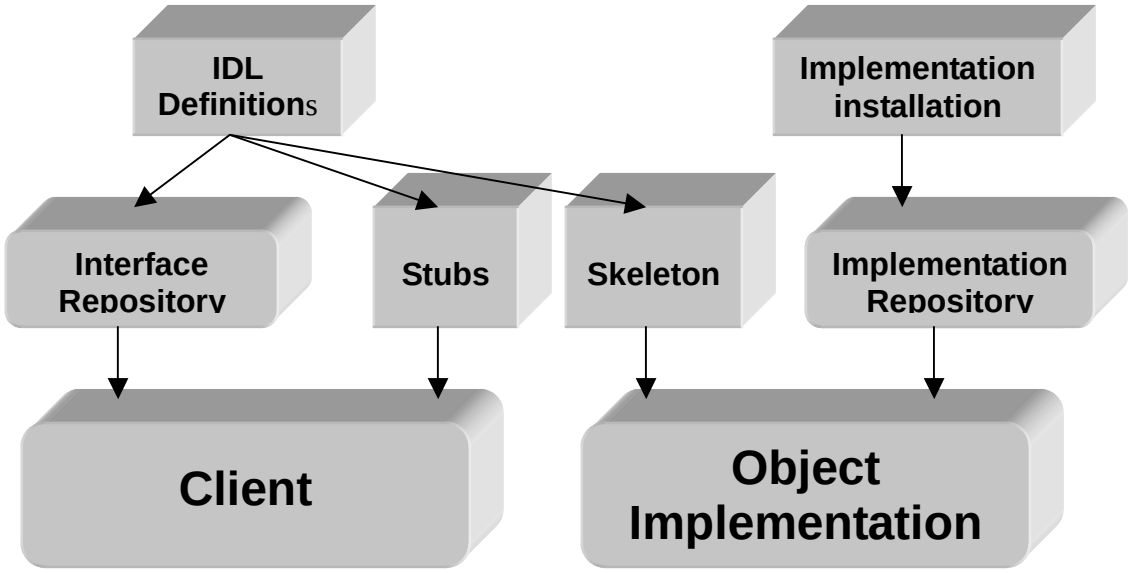
The IDL compiler generate the interface definition for use by client to invoke object on remote machine. The code generated for client is called stub code while the code generated for object implementation is called skeleton code. The client and object implementation are linked up by these code together with the ORB run-time system. They are providing static invocation in which only the interface defined at compile time can be invoked.



Actually, the interfaces to objects can be defined in two ways. Using OMG IDL, we can defined the interface statically. Alternatively, interfaces can be added to the Interface Repository service. It represents the components of an interface as objects, permitting run-time access to these components.

There are other interfaces at work in the ORB to provide other function and facilities. The are interfaces that let client to request for any operation dynamically. The interface in the client side is Dynamic Invocation Interface and the interface on the object side is Dynamic Skeleton Interface. Besides, there is interface for the client or server to talk to ORB for ORB initialization and object reference manipulation. For the object implementation, there is Object Adapter used for the management of interaction with ORB.

The figure below showed how the interface and implementation information can be available for use by clients and object implementations. The interface is defined in OMG IDL and/or in the Interface Repository. These definitions will be used for the generation of client stubs and object implementation skeletons.



The object implementation information is available at the time of installation and it will be stored in the Implementation Repository for later use.

Client Stubs

When client invoke an operation on object reference, it must link to the stub to convey the invocation to the remote object. The stubs are instantiated as local proxy objects that delegate invocations on their methods to the remote object. The stub are optimized for a particular ORB core to call the rest of ORB with this private interface. There may be different stubs correspond to different ORBs if more than one ORB is available. It is necessary for the ORB and language mapping to associate the correct stubs with particular object reference.

Dynamic Invocation Interface

An interface is available the dynamic construction of object invocation. If the client know the object reference and its interface type, it can build request to the object without prior stub code generation by IDL compiler. The client must supply information about the operation to be performed and the types of parameters passed.

Implementation Skeleton

When the request reach the server, there must be a way to invoke the method on the right object implementation. It is the skeleton code that unmarshal the code and passed the data to the object implementation.

Dynamic Skeleton Interface

This is an interface that allow dynamic handling of object invocation. The object implementation is reached via this interface analogous to the Dynamic Invocation Interface in the client side. The object implementation will provide the description of all

operation parameters to the ORB and the ORB will provides the value of input parameters for performing the operation. The object implementation will then send back the result together with any exception to the ORB to return to the client.

Object Adapters

An Object Adapter is a logical set of serve-side facilities that serves to both extend the functionality of the ORB and to provide a mechanism for the ORB and the object implementation to communicate with each other. The object implementation will use the object adapter to let the ORB access itself. The ORB can use object adapter to manage the run-time environment of the object implementation. Different object adapters can be used to offer specialized services that have been optimized for a particular environment, platform or object implementation. A typical object adapter provide the following services:

- Registration of servers
- Activation and deactivation of object implementations
- Instantiation of objects at run time and the generation and management of object references
- Mapping of object references to their implementations
- Dispatching of client requests to server objects via skeleton or Dynamic Skeleton Interface

There are two object adapters adopted by CORBA, the Basic Object Adapter (BOA) and the Portable Object Adapter (POA). They are mainly used to make and interpret object references and to activate and deactivate object implementations.

Interface Repository

The Interface Repository is a fundamental service in the CORBA. It provides a set of objects that contains the IDL definitions in a form available at run-time. It may be use by ORB at request. Through this repository, it is possible to determine the operations available at an object whose interface is not known at compile time. Hence, it make dynamic invocation possible. In addition, the Interface Repository also serve to store additional information associated with interfaces to ORB objects. For example debugging information, libraries of stubs and skeleton or routine to browse or format certain kinds of object.

Implementation Repository

The Implementation Repository is a place to store information to allow ORB to locate and activate object implementation. It is proprietary to each ORB and it stores information on installation of implementation and control policies of object activation. In addition, it store additional information associated with the implementation of object. For example, debugging information, administrative control, resource allocation and security.

Interoperability

Many different ORB products are currently available. The diversity of products allows different vendors to produce products to suit specific operational environments. In addition, there are distributed systems which are not CORBA compliant and there is a need to provide interoperability between those systems and CORBA. To deal with these, the OMG has formulated the ORB interoperability architecture.

Different objects implemented in different environment not only unable to communicate with each other. There is also security problem to solve. In order to provide a fully interoperable environment, CORBA introduce the concept of domain. It defines domains as islands within which objects are accessible because they use the same communication protocols, the same security, and the same way of identifying objects. Objects from different domain need some bridge to facilitate translation of protocol, identity and authority between domains.

The basic elements of interoperability are as follows:

- ORB interoperability architecture
- Inter-ORB bridge support
- General and Internet Inter-ORB Protocols (GIOPs and IIOPs)

There is also environment-specific inter-ORB protocols (ESIOPs) that are optimized for a particular environments.

ORB Interoperability Architecture

The ORB Interoperability Architecture provides a framework for defining different elements of interoperability and identifying the things that can be used as

common representations between domains. The architecture introduces the concepts of immediate and mediated bridging of ORB domains. The Internet inter-ORB Protocol (IIOP) which will be discussed below forms the basis for broad-scope mediated bridging. The Inter-ORB bridge support can be used to implement both immediate bridges and to build half-bridge to mediated bridge domains. With bridging techniques, different ORB can interoperate without knowing the details of the ORB's implementation.

Inter-ORB Bridge Support

When two ORB are in the same domain, they need to use bridge to communicate. The bridge will ensure that the semantics and content are mapped from one form in an ORB to the other form in another ORB. A bridge that provides one-to-one protocol translation is called a full bridge which performs immediate bridging. This is a simple and effective solution as long as the number of protocols remains small. Besides, half bridge and mediated bridging can be used to avoid increase in the number of bridge cause by increasing number of protocol need to be translated. The inter-ORB bridge support defines ORB APIs and conventions so as to enable the building of interoperability bridges between ORBs. The inter-ORB bridge support can also be used to communicate with non-CORBA system such as Microsoft's Distributed Component Object Model (DCOM).

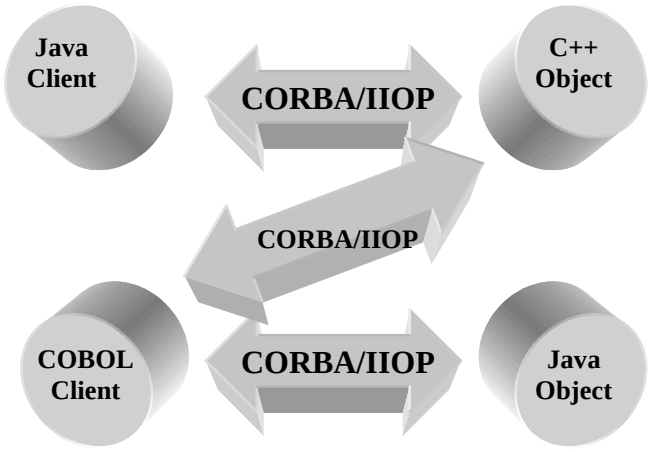
General Inter-ORB Protocol (GIOP)

In order to let different domains to use bridges, a standard transfer protocol is required. CORBA has adopted a basic inter-ORB protocol called General Inter-ORB Protocol (GIOP). The protocol is simple, scalable and easy to implement. It serves as

a common backbone protocol so that the number of different combinations of half bridges needed between domains is minimized. Th GIOP itself will not support full interoperability. Its specialized form, Internet Inter-ORB Protocol (IIOP) will do.

Internet Inter-ORB Protocol

The OMG also defined a specialization of GIOP, called the Internet Inter-ORB Protocol (IIOP) that use TCP/IP as transport layer. It is designed to provide "out of the box" interoperability with other compatible ORBs as TCP/IP is the commonest vendor independent transport layer. Furthermore, IIOP can also used in bridging two or more ORB by implementing half bridge which communicate with IIOP. Vendors can also use it for internal ORB messaging.



Environment-Specific Inter-ORB Protocols (ESIOPs)

The architecture also allow open ended set of Environment-Specific Inter-ORB Protocols (ESIOPs). This protocol will be used at user sites where a particular

networking r distributing computing infrastructure is already in general use.

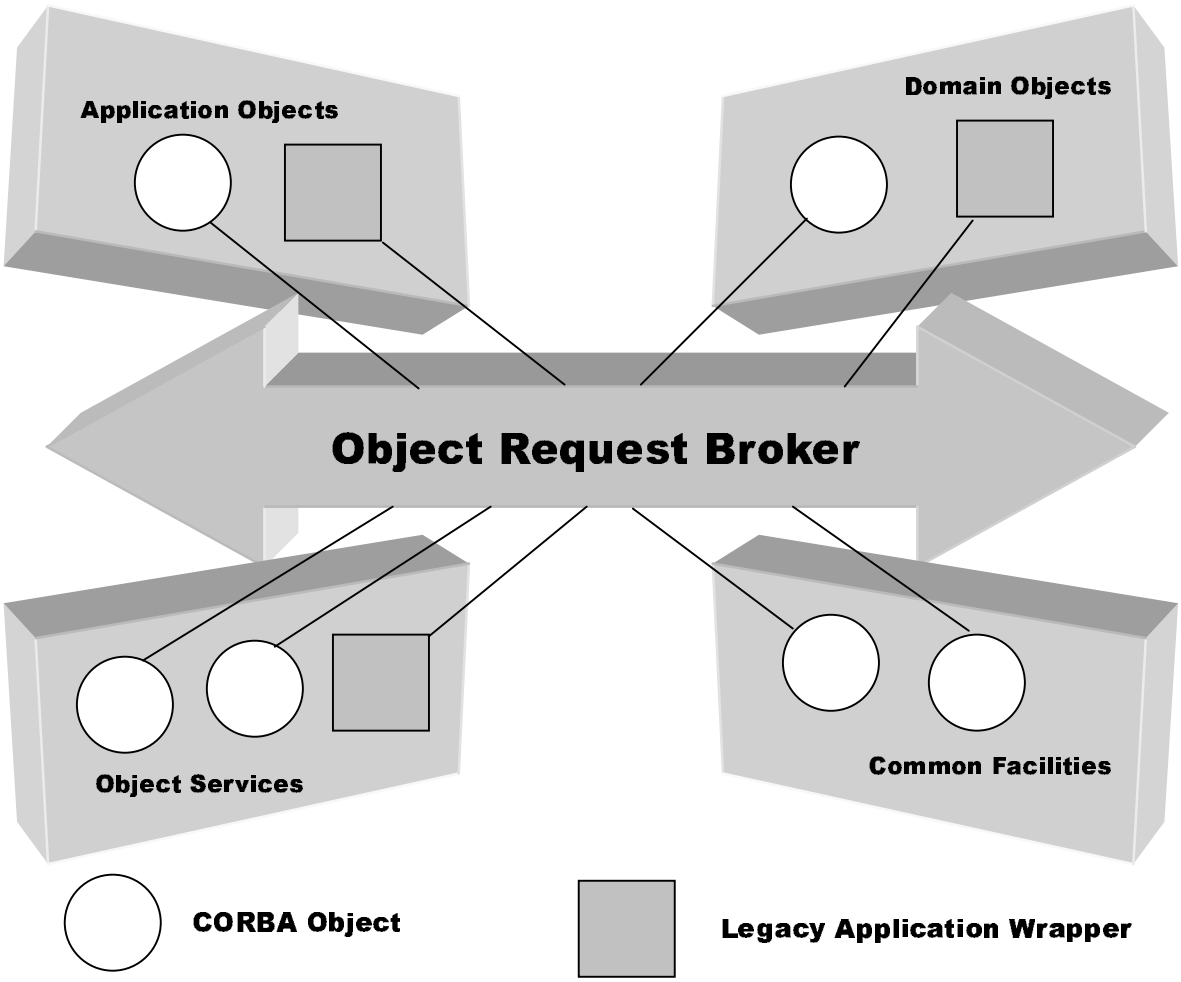
Object Management Architecture (OMA)

In the real world, distributed object computing requires much more than a communication mechanism. We need an infrastructure. Applications need to locate objects that may migrate about the network. Objects that the applications need may be dormant and require activation. Applications need to obtain services based on general property descriptions rather than specific identities. Such list of requirements will go on. They are met by a distributed computing infrastructure, an architecture of underlying mechanisms and basic services that provide a stable, powerful platform upon which applications can be built. This is what OMG Object Management Architecture (OMA) provided.

The OMA is the framework within which all OMG adopted technology will fits. CORBA only specific how objects can communicate with each other without further support such as naming service, transaction service or security. Three other components together with the ORB and application objects form the OMA Reference Model. They are:

- ✧ CORBAservices
- ✧ CORBAfacilities
- ✧ CORBAdomain

The OMA Reference model can be shown in the figure below:



- ORB: The *Object Request Broker (or ORB)* is the piping that connects all developments within the CORBA universe. It specifies how all objects written in any language running on any machine can communicate with each other.
- CORBAservice: A *CORBAservice* is a specification for some added CORBA functionality with implications in a horizontal arena. A CORBAservice will usually define some object-level functionality that you need to add to an application but do not want to produce in-house. Because all vendors producing a given

CORBAservice conform to the specifications produced by the OMG, you can easily swap one vendor's solution for another vendor's solution. An example of a CORBAservice is the event service that allows events to be delivered from one source object to a collection of listener objects.

- ✧ CORBAfacility: Like a CORBAservice, a *CORBAfacility* is also a specification for some added CORBA functionality. However, the implications can be either horizontal or vertical. A CORBAfacility differs in that it specifies functionality at a higher level than a CORBAservice. For example, CORBAfacilities define functionality for transactions such as email and printing.
- ✧ CORBAdomain: A *CORBAdomain* is a specification for some level of CORBA added functionality with applications in a unique industry or domain. A CORBAfacility used in finance might calculate derivative prices, and a CORBAfacility used in healthcare might match up patient records contained in heterogeneous systems.

CORBAservices

CORBAservices add functionality to a CORBA application at the server level. They provide services to objects that are necessary for various tasks, including event management, object lifecycle, and object persistence. New CORBAservices are constantly being produced, but at the time of this writing, only the following 15 services are in existence:

- ✧ Collection Service: This service provides access to a variety of data structures.
- ✧ Concurrency Control Service: This service enables multiple clients to coordinate access to shared resources. For example, if two clients are attempting to withdraw funds from the same back account, this service could be used to ensure that the two transactions do not happen at the same time.
- ✧ Event Service: This service enables events to be delivered from multiple event sources to multiple event listeners.
- ✧ Externalization Service: This service enables an object (or objects) or a graph to be written out as a stream of bytes.
- ✧ Licensing Service: This service enables control over intellectual property. It allows content authors to ensure that their efforts are not being used by others for profit.
- ✧ Life Cycle Service: This service defines conventions for creating, deleting, copying, and moving objects.
- ✧ Naming Service: This service allows objects to be tagged with a unique logical name. The service can be told of the existence of objects and can also be queried for registered objects.

- Persistent Object Service: This service enables objects to be stored in some medium. This medium will usually be a relational or object database, but it could be virtually anything.
- Property Service: This service enables name/value pairs to be associated with an object. For example, some image file could be tagged with name/value pairs describing its content.
- Query Service: This service enables queries to be performed against collections of objects.
- Relationship Service: This service enables the relationship between entities to be logically represented.
- Security Service: This service enables access to objects to be restricted by user or by role.
- Time Service: This service is used to obtain the current time along with the margin of error associated with that time. In general, it's not possible to get the exact time from a service due to various factors, including the time delta that occurs when messages are sent between server and client.
- Trader Object Service: This service allows objects to locate certain services by functionality. The object will first discuss with the trader service whether a particular service is available; then it negotiates access to those resources.
- Transaction Service: This service manages multiple, simultaneous transactions across a variety of environments.

CORBAservices are always being developed by the OMG. Firewall and fault tolerance services are being finalized.

CORBAfacilities

CORBAfacilities add additional functionality to an application at a level closer to the user. Facilities are similar to services in that they both aid a CORBA application; however, CORBAfacilities need not be simply targeted at a broad audience. CORBAfacilities are categorized into horizontal and vertical services.

Vertical CORBAfacilities

A vertical CORBAfacility has specific applications in a unique industry or domain. Obvious parallels exist between a vertical CORBAfacility and a CORBAdomain; however, CORBAdomains usually have much broader applications within the domain. The following list describes the eight existing vertical CORBAfacilities:

- Accounting: This facility enables commercial object transactions.
- Application Development: This facility enables communication between application development objects.
- Distributed Simulation: This facility enables communication between objects used to create simulations.
- Imagery: This facility enables interoperability between imaging devices, images, and image data.
- Information Superhighways: This facility enables multiuser application communication across wide area networks.
- Manufacturing: This facility enables interoperability between objects used in a

manufacturing environment.

- Mapping: This facility enables communication between objects used for mapping.
- Oil and Gas Industry Exploitation and Production: This facility enables communication between objects used in the petroleum market.

Horizontal CORBAfacilities

Horizontal CORBAfacilities are broad in their function and should be of use to virtually any application. Due to their broad scope, there are four categories of horizontal CORBAfacilities:

- User Interface: All facilities in this category apply to the user interface of an application.
- Information Management: All facilities in this category deal with the modeling, definition. storage, retrieval, and interchange of information.
- System Management: All facilities in this category deal with management of information systems. Facilities should be neutral in vendor support, because any system should be supported.
- Task Management: All facilities in this category deal with automation of various user- or system-level tasks.

The User Interface common facilities apply to an application's interface at many levels. As shown in the following list, this includes everything from physically rendering object to the aggregation of objects into compound documents:

- Rendering Management: This facility enables the physical display of graphical objects on any medium (screen, printer, plotter, and so forth).
- Compound Presentation Management: This facility enables the aggregation of multiple objects into a single compound document.
- User Support: This facility enables online help presentation (both general and context sensitive) and data validation.
- Desktop Management: This facility supports the variety of functions needed by the user at the desktop.
- Scripting: This facility exists to support user automation scripts.

The Information Management common facilities enable the myriad functions required in a data ownership situation. These facilities, defined in the following list, range in function from information management to information storage:

- Information Modeling: This facility supports the physical modeling of data storage systems.
- Information Storage and Retrieval: This facility enables the storage and retrieval of information.
- Compound Interchange: This facility enables the interchange of data contained in compound documents.
- Data Interchange: This facility enables the interchange of physical data.
- Information Exchange: This facility enables the interchange of information as an entire logical unit.
- Data Encoding and Representation: This facility enables document encoding

discovery and translation.

- Time Operations: This facility supports manipulation and understanding of time operations.

The System Management common facilities aid in the difficult task of managing a heterogeneous collection of information systems. These facilities, defined in the following list, range in function from managing resources to actually controlling their actions:

- Management Tools: This facility enables the interoperation of management tools and collection management tools.
- Collection Management: This facility enables control over a collection of systems.
- Control: This facility enables actual control over system resources.

The Task Management common facilities assist with the automation of user- and system-level tasks:

- Workflow: This facility enables tasks that are directly part of a work process.
- Agent: This facility supports manipulation and creation of software agents.
- Rule Management. This facility enables objects to both acquire knowledge and to also take action based on that knowledge.
- Automation: This facility allows one object to access the key functionality of another object.

CORBAdomains

CORBAdomains are solutions that target an individual industry. They differ from vertical CORBAfacilities in that they often fully model some specific business process. There are several specifications that apply to special area markets or domains. Each specialty area represents the needs of an important computing market. For example, CORBA Finance targets a vitally important vertical market: financial services and accounting. These important application areas are present in virtually all organizations: including all forms of monetary transactions, payroll, billing, and so forth. The state of the practice entails many monolithic and proprietary application systems with limited standards for data interchange and commercial software component integration. Suppliers and end-users in this market have a significant need for the interoperability and portability benefits provided by current and future OMG specifications aimed at these users. In addition to CORBA Finance, other domains include:

- CORBA Business
- CORBA Electronic Commerce
- CORBA Lifesciences
- CORBA Med
- CORBA Manufacturing
- CORBA Telecoms
- CORBA Transportation

As specifications become adopted and approved by OMG, they will be included in the CORBA domain documentation set.

Java and CORBA

Introduction

Although Java and CORBA each introduce a different approach to distributed computing, they appear to be made for each other. Java is high-level programming language that is object-oriented, platform independent and distributed. From Orfali and Harkey:

> "Java is the first step toward creating an Object Web, but it is still not enough. Java offers tremendous flexibility for distributed application development, but it currently does not support a client/server paradigm. To do this, Java needs to be augmented with a distributed object infrastructure, which is where OMG's CORBA comes into the picture. CORBA provides the missing link between the Java portable application environment and the world of intergalactic back-end services. The intersection of Java and CORBA object technologies is the natural next step in the evolution of the Object Web."

The object models of Java and CORBA correspond closely to the other. Both of them support abstract interface. The CORBA IDL data types can be easily map to Java data types. They provide very similar interface inheritance mechanisms. The mapping of CORBA name spaces modules to Java package is simple. From the architecture point of view, they are complementary to each other. Java gives you portable objects for easy distribution on the network. CORBA provides an infrastructure to connect the objects together and also integrate with other elements including databases,

legacy systems, object or applications written in other languages.

Java in CORBA Architecture

Java provide following unique features in CORBA environment:

- Portability across platforms
- Programming in Internet
- Object-oriented language
- Component model

Portability

Java is a mobile object system. It is a portable OS for running objects. Java will allow your CORBA objects to run on everything from mainframes to network computers to cellular phones. Java bytescodes allow simplified code distribution. It enable a single code to be used on any platform without porting. Hence, it helps to reduce the development and maintenance costs.

Programming in Internet

Java allows the implementation of CORBA clients as applets. Applets can be run in web browser to access CORBA objects. Mainstream browser such as Netscape Communicator has already incorporated commercial ORB to enable CORBA access.

Object-oriented programming

The IDL to Java mapping is natural and direct. Java also provides a cleaner approach to object-oriented programming than C++, with fewer memory management responsibilities, no pointers, a less confusing syntax, and simpler method resolution rules.

Its built-in multithreading, garbage collection, and error management make it easier to write robust networked objects.

Component model

Java's component model, Java Beans, allow software developer to build reusable software components. The components can be easily put together to achieve new functionality.

CORBA in Java Programming

CORBA is a lot more than ORB as it is a very complete distributed object platform. It extends the reach of Java applications across networks, languages, component boundaries, and operating systems. The CORBA's high-level distributed object paradigm provide the following advantages:

- Implementation independent interface
- Programming language independence
- Location Transparency and Server Activation
- Automatic Stub and Skeleton Code Generation
- Reuse of CORBA Services and Facilities

Implementation Independent Interfaces

The Interface Definition Language (IDL) of OMG allow the separation of the interface from the implementation of distributed object applications. This is particularly useful for the software engineering processes. Systems designs based on object-oriented design methodologies and tools can be ex pressed in OMG IDL. Once the interfaces are specified in IDL, different teams can implement different parts independently.

Programming Language Independence

The OMG has provided a no. of mapping for the IDL to other languages. Different parts of a CORBA system can be implemented in different languages. All interactions through the interfaces will be independent of the programming language they are implemented. With CORBA, legacy system can be supported. The most appropriate

programming language can be used to build objects. The legacy system can be wrapped using chosen language to form an object in the CORBA system.

Location Transparency and Server Activation

Location transparency can be provided by CORBA. Objects can be identified independently of the physical location of the object. It can potentially change its location without breaking the application. Objects can also be activated on demand. There is no need to start up all object servers before the application run.

Automatic Stub and Skeleton Code Generation

In traditional distributed system, a number of lower level and repetitious programming work such as opening, closing and controlling of network connection are required. It will also involved marshaling and unmarshaling of data and setting up servers to listen at a port. In CORBA, the IDL compilers will generate code for the data representation in a particular language. Code for marshaling and unmarshaling of data will also be generated.

Reuse of CORBA Services and Facilities

As mentioned in previous section, CORBA provide CORBAservices and CORBAfacilities. They provided functionality at server level and also close to user with different interests. They provided reusable common facilities and services.

Java, CORBA and Web

The web began as a means to distribute large amount of information in the Internet. It evolved continuously with new functionality added to provide more and more complex interactive applications. HyperText Transport Protocol (HTTP) and Common Gateway Interface (CGI) provided certain interactions in the web and it was also used to access back-end systems such as databases. However, it has some problems can drawbacks.

Current Problems in Web

Current web interactivity using CGI and HTTP is slow, stateless and cumbersome. It is not suitable for writing modern client /server applications. Although web server vendors have gone through numerous contortions to work around the limitation of CGI, their solution are usually in form of proprietary server extensions.

In typical CGI-based solution, a client has some state which is affected by the data entered into a form or as a result of state changes in the server. The client is a series of HTML pages where each page is created by CGI call to program in server. All client state has to be passed to a program behind the CGI. To get around stateless problem, some server extensions may require cookies to identify their state. These attempts are mostly proprietary.

The CGI is slow. There are a number of performance bottlenecks in the CGI-based approach. It launches a new process to service each incoming client request.

Different vendor extensions provide work-around but that would introduce more non-standard things.

The main problem with these approaches is that they require HTTP and web server to mediate between objects running on the client and on the server. There is no way for a client object to directly invoke a server object. This approaches is not suitable for full-blown client/server applications that require highly interactive conversations between components. It also does not scale well.

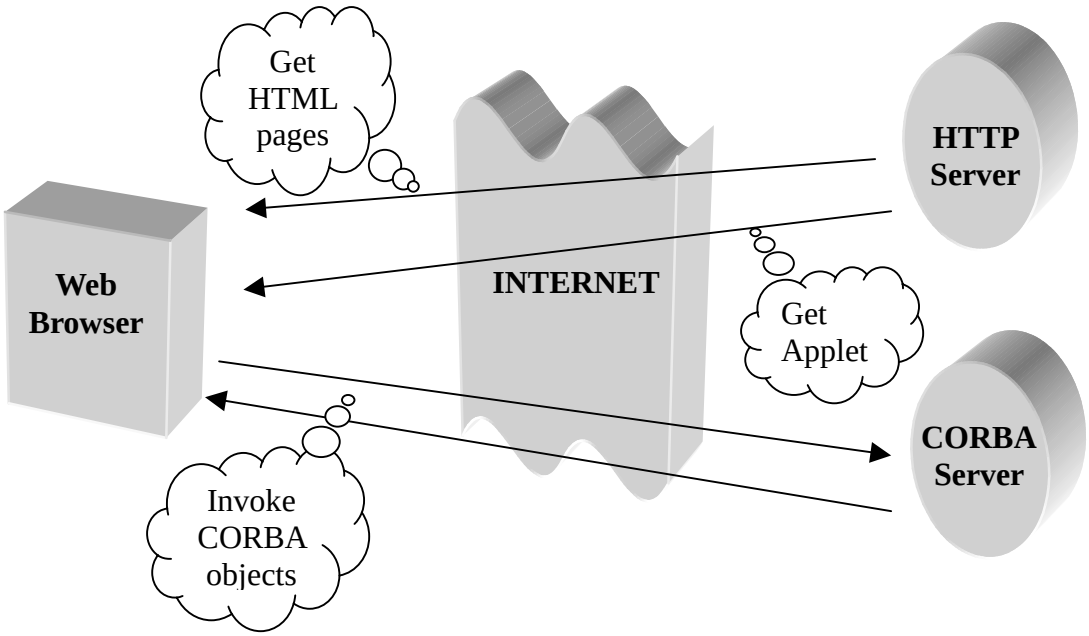
With the CORBA approaches to the web-based distributed application, Java ORBs solve the stateless problem. Client and server program are continuously executing and maintaining their own states. ORB infrastructure allows the invocation of operations on remote objects, which communicate only the data they need for each interaction. The ORB also maintains a network connection between client and server, keeping a reasonable trade-off between lowering connection establishment overhead and freeing idle network resources.

CORBA approaches to web-based applications

With CORBA, web-based client can interact with server objects as follows:

1. Web browser downloads HTML page with embedded Java applets.
2. Web browser retrieve Java applet from HTTP server.
3. Web browser run in the Java virtual machine (JVM) of web browser.
4. Applet invokes CORBA server objects. The applet will contains stub to invoke objects on server through ORB.

5. CORBA allow the interaction between applet and server object without switching out of the page. The connection between the applet and objects will persist until disconnection.



CORBA Implementation

Visibroker for Java

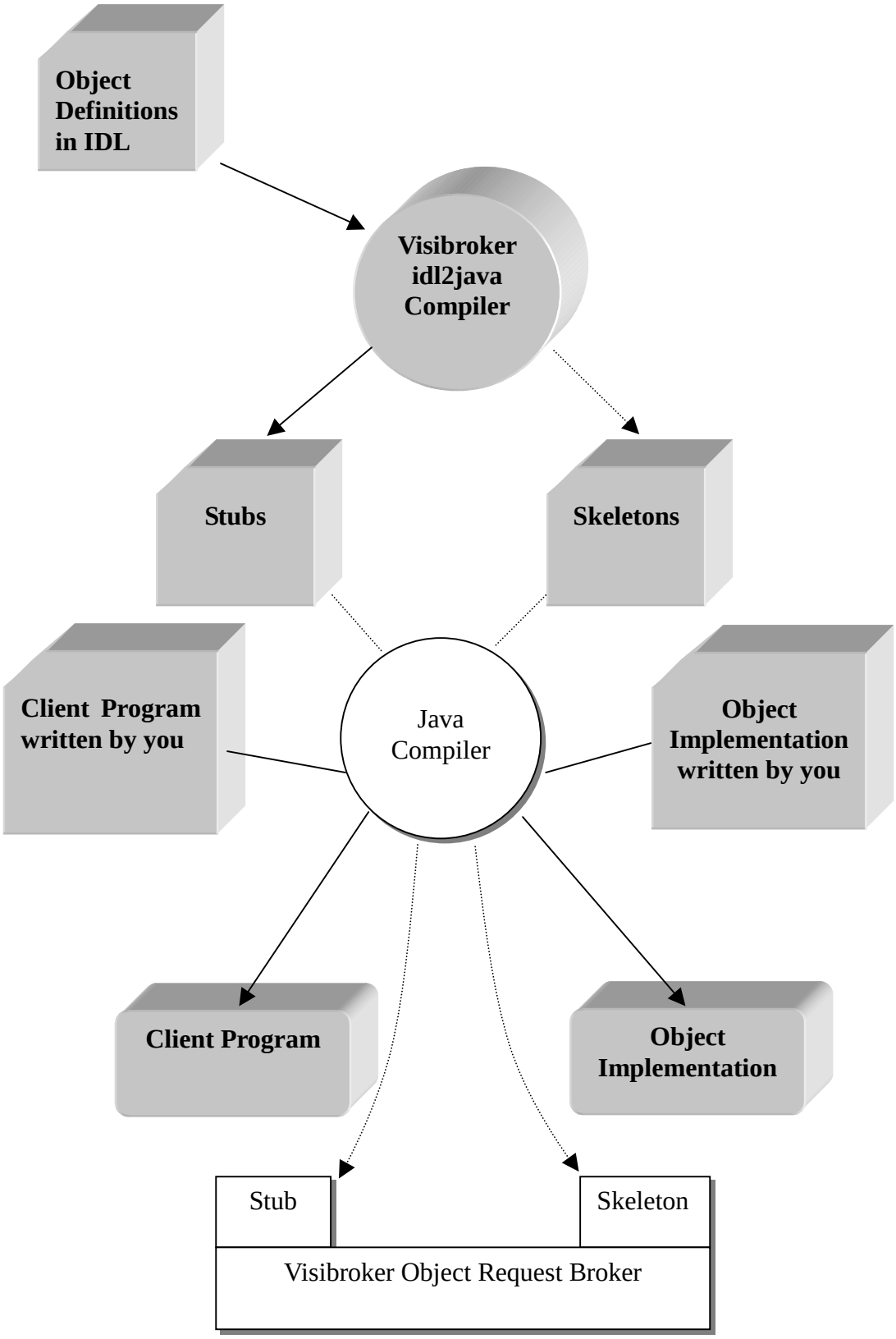
Visibroker for Java is a CORBA 2.0 Object Request Broker (ORB) and supports a development environment for building, deploying and managing distributed object applications. Objects built with Visibroker for Java are accessed by Web-based applications that communicate with Internet Inter-ORB Protocol (IIOP). I will use Visibroker for Java in the development of the applications.

Developing applications with Visibroker

The steps to create an application with Visibroker are as follows:

1. Specifying all objects and their interfaces using OMG's Interface Definition Language (IDL)
2. Using Visibroker idl2java compiler to generate stub routines for client program and skeleton code for the object implementation
3. Write client programs and use stub routines for method invocations on server objects.
4. Write servers code together with skeleton code to implement the server objects.
5. The code for the client and object, once completed, is used as input to your Java compiler to produce a Java applet or application and an object server.

The following figure illustrate the above steps:


Object Definitions in IDL
Visibroker idl2java Compiler
Stubs
Skeletons
Client Program written by you
Java Compiler
Object Implementation written by you
Client Program
Object Implementation
Stub
Skeleton
Visibroker Object Request Broker

Visibroker features

Visibroker for Java has several key features. Some of which will be used in this project. They are listed and will be explained below:

- ✧ Smart binding
- ✧ Smart Agents
- ✧ Object Activation Daemon
- ✧ Gatekeeper

Smart binding

Visibroker enhances performance by choosing the optimum transport mechanism whenever a client binds to a server object. If the object is local to the client process, the client performs a local method call. If the object resides in a different process, the client uses IIOP.

Smart Agents

The Visibroker Smart Agents is a simplified naming service that provides a bootstrapping object discovery mechanism for client. The smart agent also provides some fault-tolerance and load-balancing facilities. A smart agents can automatically reconnects a client application to an appropriate object server if the server currently being used becomes unavailable due to a failure. Furthermore, Smart Agents can use Visibroker ‘s Object Activation Daemon to launch instances of a server process on demand.

Object Activation Daemon

To automatically activate a server when a client requests a bind to the object, you can register an object implementation with Visibroker's Object Activation Daemon (OAD). The OAD includes command-line utilities for registering, unregistering, and listing objects or you can use methods available from the OAD interface.

Gatekeeper

While still conforming to the security restrictions imposed by web browsers, the Visibroker Gatekeeper runs on a web server and enables client programs to make calls to objects that do not reside on the web server and to receive callbacks.

System Specification

Introduction

In this project, we will implement a prototype of the web-based learning system in distributed object architectures based on Common Object Request Broker Architecture (CORBA). Client and server programs will be implemented and academics, educational administrators or students can use the client program for learning, institute administration or course management.

A distributed object architecture would creates objects that correspond to the primary education business elements. These will include entities such as “student”, “course” or “library”. The clients will be either Java applets in web pages or standalone Java applications that access objects through an Object Request Broker, which locates the desired object wherever it may reside in the network. The Visibroker Object Request broker and other features including smart agents, object activation daemon and Naming Service will be used.

Web-based Learning System Object Model

A initial set of objects that may be of interest in supporting the web-based learning system is as follows:

- ✧ Students (contains student ID, password, student name, sex, address, telephone no., programme of study, course taking, course completed)
- ✧ Tutors (contains staff ID, password, staff name, sex, address, telephone, course

tutoring)

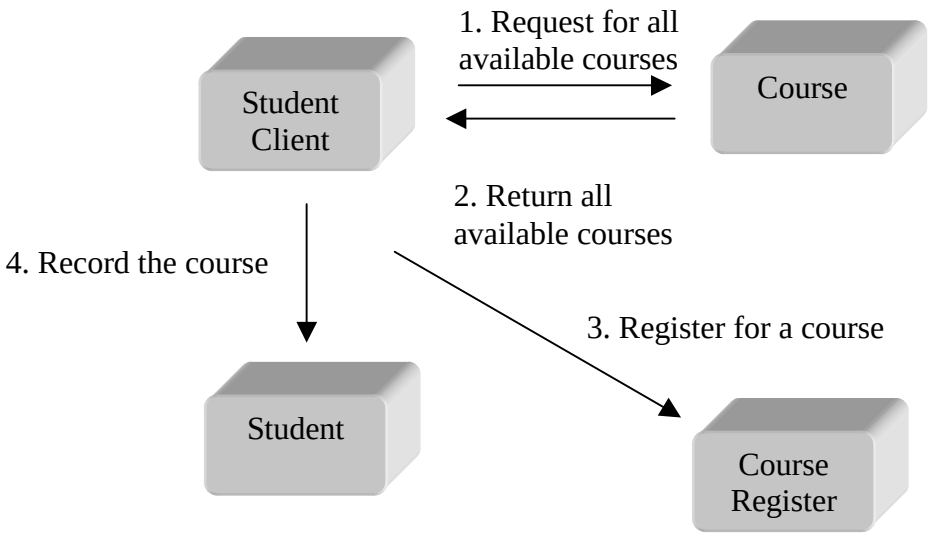
- Lecturer (staff ID, password, staff name, sex, address, telephone no., course managing)
- Administrators (staff ID, password, staff name, sex, address, telephone no.)
- Librarian (staff ID, password, staff name, sex, address, telephone no.)
- Course (course code, course name, school, course link, resourceList, assignList, credit)
- Course Register (student ID, course code, course result, course status)
- Assignment (assignment ID, course code, assignment link, submission deadline)
- Assignment Register (assignment ID, student ID, assignment score)
- Resources (resource ID, resource link)

A three-tiered architecture will be used. The client accessed the object implementations in the server. The object implementations are Java class that implements the operations corresponding to an Interface Definition Language (IDL) interface. The server objects will then accessed the database via Java Database Connectivity (JDBC) driver. The database will be resided in a different server away from the server. The object layer provides the necessary location transparency and isolates the application from the actual data sources. The database will be Oracle database to provide permanent storage for object's data.

Details of possible requests to objects by different types of users will be discussed in the following sections.

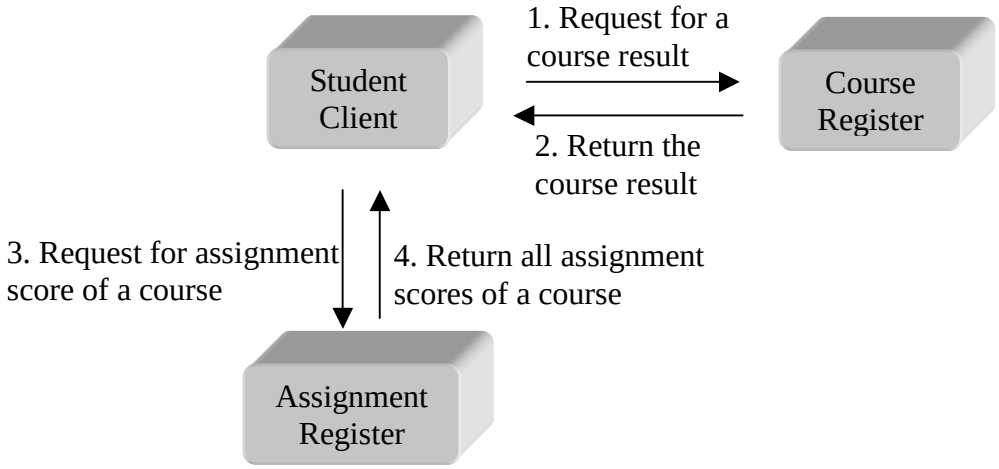
Students

In the normal workflow, student needs to have an account which is created by the administrator. The student could then login the client program using his account username and password. Student can then check the course available and then register himself to study different courses.



A collaboration diagram illustrates the registration of course by a student

At no time may a student register with courses of a total of more than 60 credits. He could then start the course through the client program and link to the course homepage. At anytime during his course of study, he may wish to withdraw from a course and he can do so via the client program. The client program also allow him to check his course results and also the score of his assignments.



A collaboration diagram illustrates the listing of course and assignment result

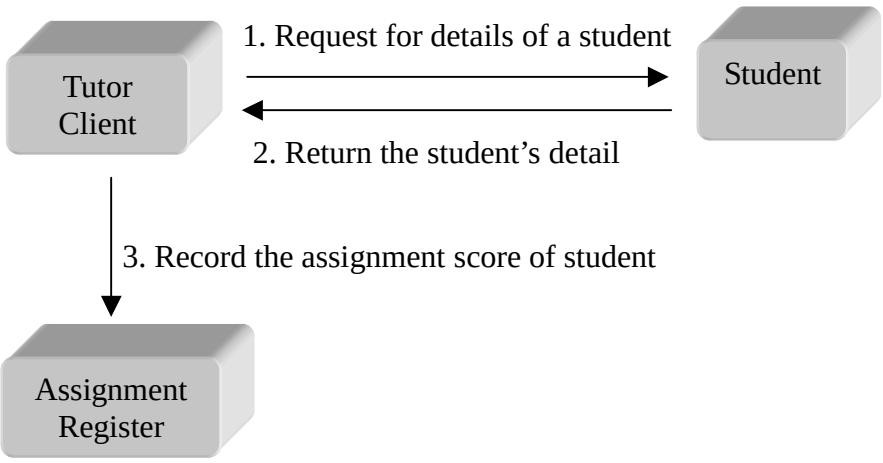
Students client program is allowed to perform the following requests:

1. Login: A student need to provide student ID and password for authentication before other requests are performed.
2. List_course: A student can request to display all the courses available for taking. The list will be sorted by course code.
3. Register_course: A student can registered for any course currently available. At no time can a student registered for more than courses with more than 60 credits. Exception will be raised if these conditions are violated
4. Withdraw_course: A student can withdraw from a course.
5. Retrieve_course_result: Students can list all his own course result for viewing
6. Retrieve_assign_score: Students can enter course code and retrieve the score of his own assignment for viewing
7. Print_course_result: Retrieve and print all his own course result.
8. Print_assign_score: Retrieve and print all his own assignment score for a particular course

9. Start_course: start the course by retrieving the web pages corresponding to the course link in course object.
10. Search_resource: Students can enter a course ID to list all the resources available.

Tutor

Tutor is responsible for the marking of the assignment of student. He can login the system and use the client program to create an assignment record of student. He can enter the score of each student or amend score of each student. A tutor can also use the client to search for the details of a student. Tutor may sometimes need to have a list of students he is tutoring. He can use the client to list the students and also print the list.



A collaboration diagram illustrates the record of assignment score of student

Tutor client program is allowed to perform the following requests:

1. Login: Tutor need to provide staff ID and password for authentication before other requests are performed.

2. Create_assignregister: Tutor can create assignment record for a particular student in the Assignment Register.
3. Amend_assignregister: Tutor can amend assignment record for a particular student in the Assignment Register.
4. Find_student: Tutor can enter the student ID to retrieve the details of a particular student
5. List_student: Tutor can list all the students that he is tutoring.
6. Print_student: Tutor can print a list of students that he is tutoring.

Lecturer

Lecturer is responsible for the course management. He can create a course so that students can register for it. Lecturer may also remove a course from future registration. He can also use the client to create course record for a student. He can enter the course result for a student or amend the result.

Lecturer client program is allowed to perform the following requests:

1. Login: Lecturer need to provide staff ID and password for authentication before other requests are performed.
2. Create_course: Lecturer can create course by provide all the details of the course
3. Remove_course: Lecturer can remove a course that he is managing.
4. Create_course_register: Lecturer can create course register for a student
5. Amend_course_register: Lecturer can amend course result in course register.

Administrator

Administrator is responsible for the overall management of the learning institute. He can create account for student using client program. Administrator can also use client to amend the details of the student's account or remove an account. Administartor may also use the client for tutor and lecturer to do the administration work.

Administrator client program is allowed to perform the following requests:

1. Login: Administrator needs to provide staff ID and password for authentication before other requests are performed.
2. Create_account: Administrator can create an account for a student
3. Delete_account: Administrator can delete an account of a student
4. Amend_account: Administrator can amend the details of a student

Librarian

Librarian client program is allowed to perform the following requests:

1. Login: Librarian needs to provide staff ID and password for authentication before other requests are performed.
2. Create_resource: Librarian can create resource for a course
3. Amend_resource: Librarian can amend resource for a course
4. Delete_resource: Librarian can amend resource for a course

Implementation Design

In order to implement the CORBA servers and clients, we must first identify the objects that will be used in the distributed object system. Then we write the IDL specification for the objects identified.

The application developed allows the management of a course with a number of students registered. The client developed is an applet and uses the server to store all information pertaining to course management. The complete IDL for the application is as follows:

IDL Specification

```
module courseServer {
    exception NoSuchUserException { string reason; };
    exception UserIDExistsException { string reason; };
    enum Degree { BA, BBA, BSC, BED };

    interface StudentI {
        attribute string     sStNo;
        attribute string     sStudentName;
        attribute string     sNotes;
        attribute float      iCredit;

        attribute Degree   type;
    };
    typedef sequence<StudentI>StudentSequence;

    struct StudentQueryS {
```

```
    string     sStNo;
    string     sStudentName;
    float      iCredit;
    Degree   type;
};

interface CourseI {
    attribute string sUserName;
    attribute string sPassword;

    StudentSequence getAllStudents();
    StudentSequence getAllStudentsByStNo();
    StudentSequence getAllStudentsByStudentName();
    void addStudent(in StudentI student);
    void deleteStudent(in StudentI student);


    StudentI obtainEmptyStudent();
};

interface RequestorI {
    void studentFound(in StudentSequence student);
};

interface CourseServerI {
    CourseI obtainCourse(in string sUserName, in string sPassword)
    raises(NoSuchUserException);

    CourseI createCourse(in string sUserName, in string sPassword)
    raises(UserIDExistsException);

    void logOut(in CourseI course);

    StudentQueryS obtainEmptyQuery();
    void searchCatalog(in StudentQueryS query, in RequestorI requestor);
```

```
        void saveCourse();
    };
};
```

From the above IDL specification, we can see the basic operation of the client and server. In the interface CourseServerI, user can login the server with the obtainCourse() operation or user can creates a new account using the createCourse() operation. All the things ends when the user call the logOut() operation. The interface also include an operation called saveCollection(). This operation save all the user information to a file so that they can be reloaded after the server restart. In order to perform a search on the student information, a searchCatalog() operation is also included. The obtainEmptyQuery () operation is an operation to return a StudentQueryS object with default values.

On the other hand, we also have CourseI and StudentI interface. The StudentI interface model on the object student while CourseI interface represent a collection of student in a course. The StudentI interface includes the basic attributes of a student object. It includes “Student Name”, “Student No.”, “Credit accumulated”, “Degree Type” and “Notes”. The CourseI interface have several operations on the student objects. It allows the addition and removal of students. Operations that caused the return of all student objects according to the attributes “StudentName” or “StNo”. The CourseI interface also include username and password attributes for the user authentication on the log in process of course management.

Design Pattern

This ReqeustorI interface is using a Callback Design Pattern. Design patterns are a tool by which the knowledge of how to build something is shared from one software engineer to another. More precisely, they allow for a logical description of a solution to common software problem. A pattern should have applications in multiple environments, and be broad enough to allow for customization upon implementation. For example, memory management in a distributed environment is tricky. Instead of inventing a new solution for every project, many developers look to the reference counting pattern as a guide.

When working with new technologies, like distributed objects, the ability to share knowledge thorough design patterns is critical. Distributed applications introduce concerns beyond those present in standalone applications, and developers new to their use will benefit greatly from any help. These concerns, including network traffic, server scalability, and general reliability, can mean project failure if neglected.

In Callback Pattern, the role of a server object is often to perform some business logic that cannot be performed by a client object. Assuming that this processing takes a significant time to perform, a client may not be able to simply wait for a server request method to complete. As an alternative, the server object can immediately return void from a request method, perform the business calculations in a unique thread, and then pass the results to the client when ready.

The theme problem of this pattern is that it is common for a client object to request some data from a server object. Assuming that the processing only takes a second or two, the client need not concern itself with the processing time involved. If, however, the server processing will take a long time, the client could end up waiting too long for a method return value. Having the client wait for this return value too long may cause client threads to hang and block, which is obviously not a desirable situation. Additionally, depending on the technology used to enable distributed computing, a timeout could occur if the server object takes too long before returning a value.

The callback pattern functions by allowing the client to issue a server request and then having the server immediately return without actually processing the request. The server object then processes the request and passes the results to the client. In most situations, the server performs all processing in a separate thread to allow additional incoming connections to be accepted.

Having Callback Pattern explained, we go back to the RequestorI interface. It is intended to model the feedback by the server when search on student information is performed. Owing to the normal delay in the search operation, the server cannot immediately deliver the required information to the client. There should be a way to notify the client when the result of search is ready. This interface allows the design in the client to deliver the required search result in a delayed manner.

In general, the IDL specification is a contract between the different distributed objects. The objects on the client side make use of the operations on the objects of the

server that is defined in the IDL. With the help of IDL to Java compiler, the IDL specification can be transformed in a number of different Java classes. We can then go on to develop our application on the server side and client side.

Implementation of Server

The implementation of server is more complicated because it involved most operations defined in the IDL specification. Actually, all the main functionality is provided by the server to the client side.

CourseServer Class

We begin the discussion with CourseServer class. We have a look at the searchCatalog() method:

```
public void searchCatalog(StudentQueryS query, RequestorI requestor) {
    StudentSearcher searcher = new StudentSearcher(query, requestor, _boa);
    searcher.start();
}
```

We will notice that even though it performs a search, the results of that search are not immediately returned. Because the search could take a long time to perform, the method returns immediately and spawns off a new thread to perform the search and then notify the client when the results are ready. To facilitate notification, the searchCatolog() method accepts a reference to the client in the form of a RequestorI object.

We should also need to take a look at the logOut() method:

```
public void logOut(CourseI course) {
        Deactivator deactivator = new Deactivator(course, _boa);
        deactivator.start();
}
```

This method also spawns off a new thread. However, this thread is charged with deactivating all objects activated by the client during a session. Ever call to BOA.obj_is_ready() needs to be paired with a call to BOA.deactivate_obj(). Not deactivating objects will lead to the server running out of memory after being used for a while.

In the CourseServer class, it also used another class called Deactivator class which deactivate objects upon logout.

```
public Deactivator(CourseI course, BOA boa) {
    _course = course;
    _boa = boa;
}

public void run() {
    ((Course)_course).deactivateObjects();
    _boa.deactivate_obj(_course);
```

This class first ask the target Course object to deactivate all activated StudentI

objects and then deactivates the Course object itself.

Next we go on to the two class Student class and Course class. They implement the StudentI interface and CourseI interface respectively.

Student Class

In the Student class, it mode a unique student with all basic properties. It helps to get or set the attributes associated with student.

```
public Student(String sStNo,
               String sStudentName,
               String sNotes,
               float iCredit,
               Degree type) {
    _sStNo = sStNo;
    _sStudentName = sStudentName;
    _sNotes = sNotes;
    _iCredit = iCredit;
    _type = type;
}
```

Course Class

We take a look at the Course class, we will see the steps taken to support

serialization. Because Course objects are going to be serialized when the server saves all objects, this class need to implement the java.io.Serializable interface. In addition to implementing the serialization tagging interface, the class also marks its BOA reference as transient. A transient object is one that is not serialized when the rest of the object is. The fact that the variable is saved, however, the information pointed to by that variable is lost.

When we are serializing objects that reference any sort of remote object, all remote references must be tagged as transient. This step required due to the fact that a remote object reference is only a pointer to an implementation object, not the implementation object itself. If the reference is serialized, there is no guarantee that the item it point to will still be there after deserialization. Because the BOA reference is needed, a method called updateTransientData() is provided, with the understanding that it will be passed a new BOA reference immediately following deserialization.

Another thing of the Course class that we need to focus on is the manner in which activated objects are tracked and then deactivated. If we look at the obtainEmptyStudent() method, we will notice that it pairs every call to obj_is_ready() with a line that places the newly activated object inside the Vector object pointed to by the _vecActivatedObjects variable. The obtainEmptyStudent() method is invoked by the client when it wants to obtain an empty StudentI object that will be added to class.

Next, we se the deactivateObjects() method. This method iterates through the set of activated objects and individually deactivates each one. The class also sets a

Boolean member variable called _bObjectsDeactivated to true, indicating that the objects have, in fact, been deactivated. This boolean is referenced in the getAllStudent() method to determine whether the StudentI objects need to be activated before being returned. Because the StudentI objects are deactivated during logout, they must be reactivated before being served again. An alternative to activating the objects in the get All Students() method would be to activate them immediately following login.

```
/**
  * Deactivates all activated objects
  */
public void deactivateObjects() {
        _bObjectsDeactivated = true;
        Enumeration e = _vecStudents.elements();
        while(e.hasMoreElements()) {
                _boa.deactivate_obj((org.omg.CORBA.Object)e.nextElement());
        }
}
```

It is designed to place the activation code in this method so as to not to slow down the login process. Users often expect certain operations to take longer than others, and applications should be designed around these expectations. Although no overall speed is gained by placing the activation code in the getAll Student() method, a perceived gain exists. Because users often expect searches to take longer than the login process, taking the time to activate objects during the search is something the user expects. If,

however, we were to activate the objects during the login, the user might be surprised how long it takes for a login to occur.

StudentSorter Class

In the Course class, we use a StudentSorter class to perform the sorting on a number of student. This class use bubble sort to sort the array of StudentI objects by either student name or student number.

```
public static void sortBySt No(StudentI[] students) {
        int iLength = students.length;
        iLength--;
        boolean bSwapHappened = true;
        while(bSwapHappened) {
            bSwapHappened = false;
            for(int i=0; i<iLength; i++) {
                if(students[i].sStNo().charAt(0) > students[i+1].sStNo().charAt(0)) {
                    bSwapHappened = true;
                    StudentI temp = students[i];
                    students[i] = students[i+1];
                    students[i+1] = temp;
                }
            }
        }
```

```
}
```

CourseHolder Class

In the CourseServer class we seen previously, we use another class called CourseHolder class to help to manage the Course objects. In managing the collection of Course objects, the CourseHolder object gets to perform a lot of interesting operations. Looking first at the Hashtable object in which Course objects are stored, note the readInHash() and saveCourse() methods.

The saveCourse() method is invoked to trigger the serialization of all Course objects. The method creates a FileOutputStream object pointing at a file titled ysers.ser and then creates an ObjectOutputStrea object on top of the FileOutputStream object. Once the ObjectOutputstream object is created, its writeObject() method is invoked with the target Hashtable object as a parameter. What we should also note about this method is that during the time that the Hashtable object is being serialized, we lock access to it using the synchronized keyword. In a multithreaded environment, it is very possible that more than one thread might attempt to access the same physical variable at the same time. Regarding this situation, that could mean a new Course object might be added to the Hashtable object at the same time a save is occuring. This situation is not one we would want to happen because it could invalidate the integrity of the Hastable's collection.

Once the contents of the Hashtable have been saved by the saveCollection()

method, it becomes possible to read them back in. The readInHash() method is invoked from the constructor and attempts to perform the deserialization. First off, the method creates a new File object that points to the users.ser file on the hard drive. The method then checks to see whether this file actually exists and simply returns a new Hashtable object if the file does not exist. If the file does exist, the method creates a new FileInputStream object on top of the File object and an ObjectInputstream object on top of the FileInputstream object. The readObject() method in the ObjectInputStream object is now invoked, and its return value is cast as a Hashtable object. Finally, we invoke the updateTransientData () method, which updates the BOA reference in each Course object.

Although the object serialization is the most complicated task performed by the Courseholder object, it is not the only task that deserves our attention. In the class, there is obtainCourse(), addCourse() and doesUserName Exist() method which provide searching for Course objects by user name and password, the addition of a ew course object and checking the existence of user name respectively.

Implementation of Client

As we have seen the implementation of the server, it is now the turn of client. The client side is usually charged with interacting with human user, gathering input and passing off data to the server for processing. The server, in turn, is charged with executing business logic, interacting with persistence mechanisms such as databases, flat files, and any processing that is too time-consuming to be performed by the client.

In the client side, there are a number of classes implemented to bring about the communication between the human user and the server. All these classes will provide the user interface by making use of the Abstract Windowing Toolkit of Java Development Kit to build the screens. These classes and their functions are list below:

Class Name	***Main Function***
CourseViewer	It extends java.applet.Applet, binds to remote services, and builds the initial user interface.
StudentDisplay	Displays a unique StudentI object and allows for modification of its values. It also prompts for user input before executing a search.
DiaplayPanel	Displays all available StudentI objects, allows for the creation of new objects, and deletes objects from the collection.
ResultsDisplay	Displays the results after a catalog search and prompts the user to add any students in the result set to his course.
DisplayMaster	Aids in the display of multiple StudentI objects.

ErrorDialog	Used to display error messages.
LoginPanel	Allows a user to either log in or create a new account.

CourseViewer Class

We will first take a look at the CourseViewer class. This class is the first class load into the page for the applet to display. It actually extends the Applet class and is charged with setting up the default look of our client. With the exception of the establishServiceReferences() method, all code simply performs the manipulation of user interface. In the establishServiceReferences() method, we obtain references to the CourseServerI object implemented in the server along with OB and BOA. The ORB object is simply used to bind to the CourseServerI object and therefore is only needed in the scope of the method itself. The BOA object, however, is needed later when registering callbacks.

As the CourseServer class was explained in the implementation of server, the searchCatalog() method accepts a RequestorI instance and notifies this object when the catalog search is complete. Because the RequestorI object reference is passed through the ORB, it is necessary to use the BOA at the client side to register the RequestorI object with the ORB. This will be explained when we discussed the DisplayPanel and ResultDisplay classes.

In addition to taking note of the manner in which remote object references are

managed, we shift our attention to note the general workflow of the client. Upon loading, the applet displays a LoginPanel object on the screen. This class prompts the user to either log in or create a new account. After receiving a command from the LoginPanel object, the applet either displays the user's course information or an error message, indicating an incorrect login or in-use login ID. All changes to the active display are not made directly through the Courseviewer object, but rather are made by requesting a change from the DisplayMaster object.

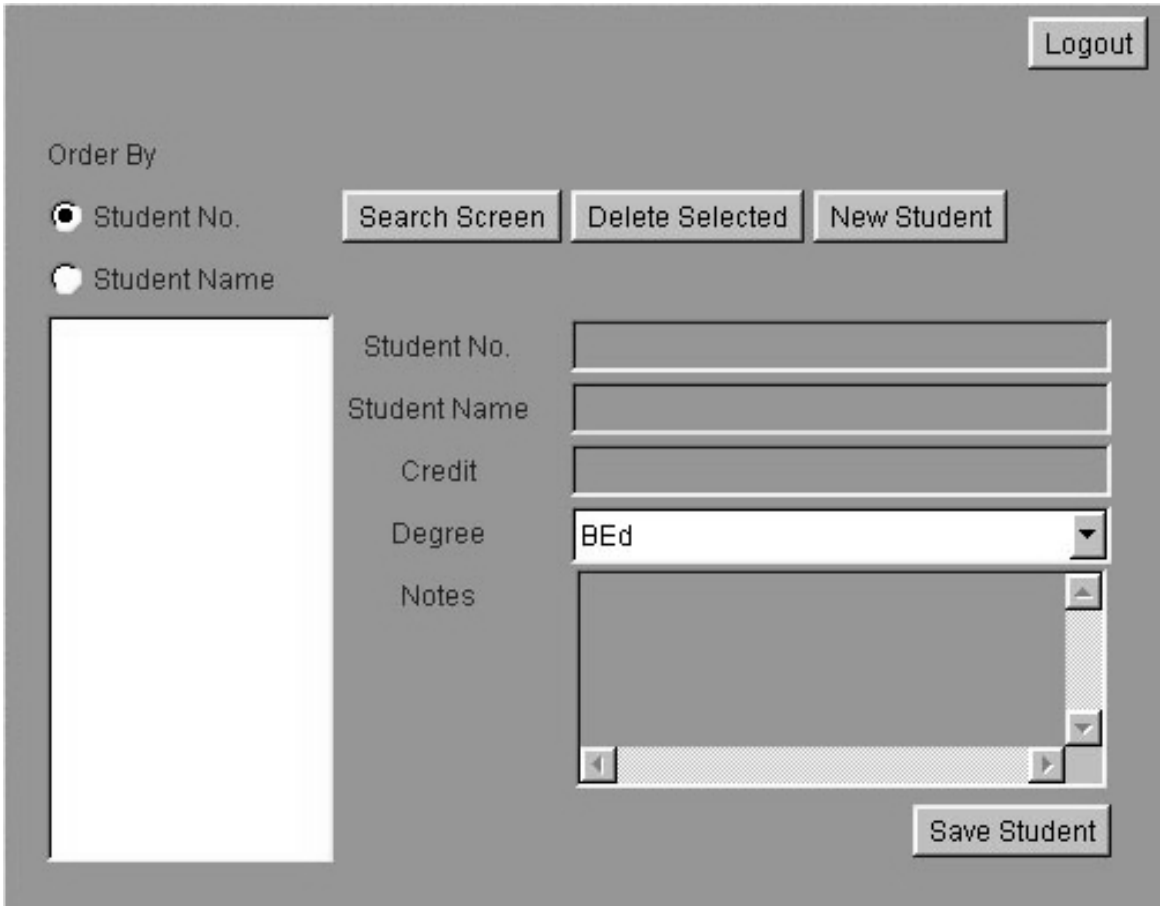
StudentDisplay Class

The next class we are going to discuss is StudentDisplay class. This class serves to either display information on a student, to collect student related information for a search, or to collect student related information used when creating a new student.

Take a look at the code of StudentDisplay class. The class uses the GridBagLayout layout manager to design the screen. This layout manager allows for advanced user interface design but has the unfortunate downside of leading to a lot of code.

```
GridBagLayout gbl = new GridBagLayout();
GridBagConstraints gbc = new GridBagConstraints();
setLayout(gbl);
Label lblStNo = new Label("Student No.");
gbc.gridx = 0;
```

```
gbc.gridy = 0;

gbc.gridwidth = 1;

gbc.gridheight = 1;

gbc.anchor = GridBagConstraints.NORTH;

gbc.fill = GridBagConstraints.NONE;

gbc.insets = new Insets(2,2,2,2);

gbl.setConstraints(lblStNo, gbc);

add(lblStNo);
```



As we have stated earlier, the StudentDisplay class serves a series of different purposes. However, each object may serve only one purpose. The unique purpose of a

single StudentDisplay object is defined by passing any one of four constants into the object's constructor during the instantiation process. These constants, shown below, allow for the object to display an StduentI object as View only or View and modify. In addition, the constants allow for the collection of student related information that is used either in a search or during the creation of a new StudentI object, which is often added to the course.

```
public static int        VIEW_ONLY = 10;
public static int        VIEW_AND_MODIFY = 13;
public static int        CREATE_NEW = 1013;
public static int        SEARCH = 42;
```

Because the value of the mode variable has an effect on the user interface, some runtime user interface decisions are made. These decisions, all made at the end of the constructor code, affect items such as the addition of a Save or Search button and the disabling of certain input fields. As is logical, View only mode allows no entry at all in any field. The Search mode disallows entry in the notes field.

During the design of the user interface, we can have two choices to deal with the notes field for search mode. One way is to simply remove the field from the screen altogether. However this could lead to user confusion. When looking at a user interface, users like consistency because it helps them to recognize the purpose of a widget without having to read its accompanying label. Through location recognition, users are able to use applications much faster than if they have to constantly figure out

where a desired widget is. By keeping the notes field on the screen and simply disabling it, we maintain the same look whenever student information is collected or displayed.

Moving down toward the bottom of the code listing, take note of the actionPerformed() method. This method is invoked when either the Save or Search button is pressed.

```
public void actionPerformed(ActionEvent ae) {
    if(_iMode == CREATE_NEW) doSaveNew();
    else if(_iMode == SEARCH) doSearch();
    else doSaveChanges();
}
```

This method take into account the active mode to determine a course of action. If the object is in Create mode, the button press is a request from the user to collect information in the user interface, pack them as a StudentI object and add it to the active course. If the object is in search mode, the button press is a request from the user to perform a search. If the user is in view and modify mode, the button press is a request from the user to save changes to the StudentI object currently being modified.

Next method of interest is the doSaveNew method which is invoked in create mode. This method first obtains a reference to a remote StudentI object from the active course. As we have discussed in the server implementation, obtaining a new StudentI object involves a call to BOA.obj_is_ready(), which means that at some point a call to

BOA.deactivate_obj() is needed. When implementing the server, we placed the code to track obj_is_ready() calls there, which means we are freed from having to do it at the client. After a reference to the new StudentI object is obtained, its attributes are set using the data entered into the user interface and finally StudentI object is added to the user's information. The last thing is to inform the user interface that a new StduentI object has been added and that the display should update itself.

```
private void doSaveNew() {
StudentI student = _course.obtainEmptyStudent();
student.sStNo(_txtStNo.getText());
student.sStudentName(_txtStudentName.getText());
try{       student.iCredit(
                    new Float(_txtCredit.getText()).floatValue()); }
catch(   NumberFormatException nfe) {
             student.iCredit(0f);
}
student.type(getDegree());
student.sNotes(_txtNotes.getText());
_course.addStudent(student);
_displayPanel.newStudent(student);
}
```

The next method that might get called when a button is pressed is the doSearch() method, which is called when a search action is to begin. This method collects all

information entered into the user interface and asks the DisplayPanel instance to perform a search.

```
        private void doSearch() {
        float iCredit = 0f;
        try{        iCredit = new Float(_txtCredit.getText()).floatValue(); }
        catch(    NumberFormatException nfe) { }
        _displayPanel.doSearch(_txtStudentName.getText(), _txtStNo.getText(), iCredit,
        getDegree());
    }
```

The last method that might be called in response to a button press is the doSaveChanges() method which is called when an StudentI object has been modified at the user interface and needs its server values updated. This method is interesting in that it only interacts with the active StudentI object itself. Because that StudentI object is only a reference to an implementation object sitting at the server, invoking any of its set methods immediately reflects the change at the client and at the server.

```
    private void doSaveChanges() {
        _student.sStNo(_txtStNo.getText());
        _student.sStudentName(_txtStudentName.getText());
        try{        _student.iCredit(
                            new Float(_txtCredit.getText()).floatValue()); }
        catch(    NumberFormatException nfe) {
```

```
                _student.iCredit(0f);
        }
        _student.type(getDegree());
        _student.sNotes(_txtNotes.getText());
    }
```

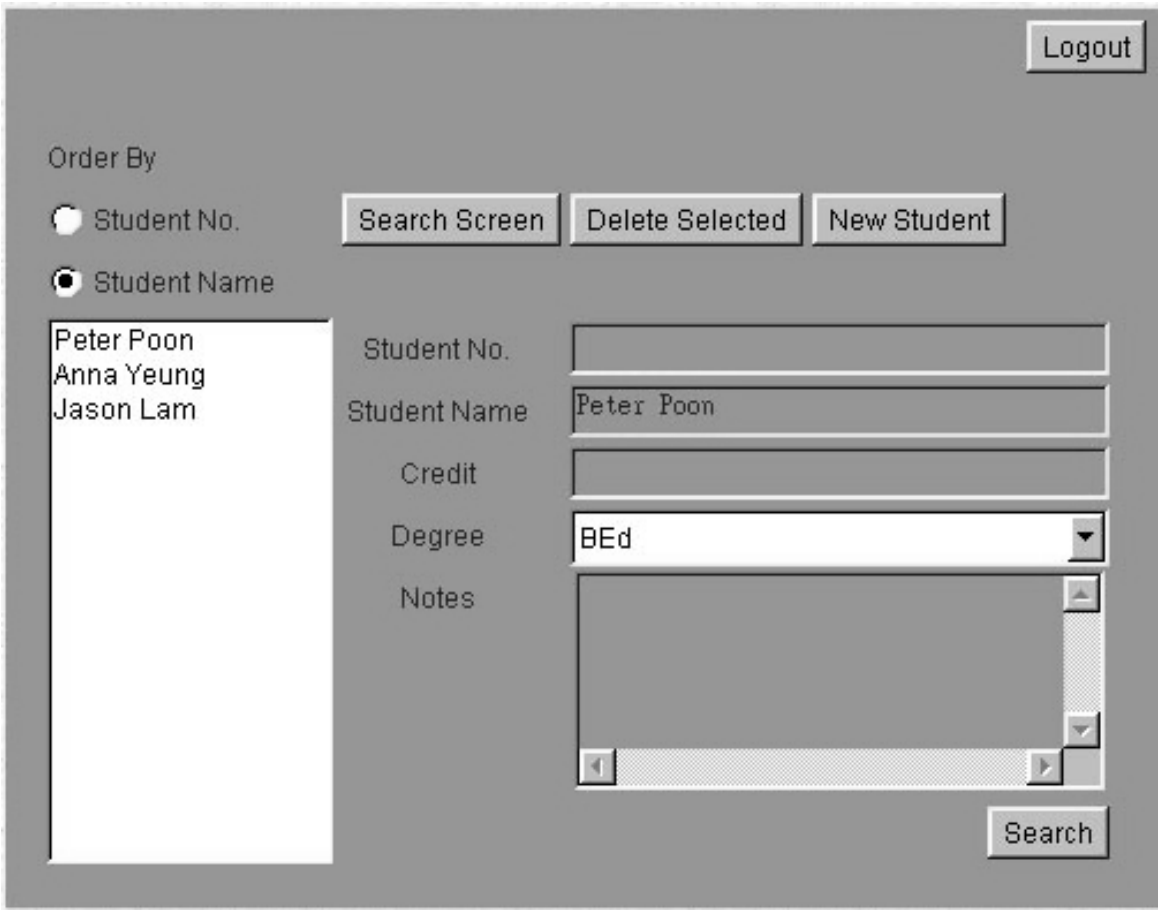
DiaplayPanel Class

The next class to be discussed is the DisplayPanel class. It creates the main user interface for user to interact with the application. The codes uses the GridBagLayout layout manager to place a List object displaying the collection of students along the left, and it places any one of many StudentDisplay objects along the right side of the screen. Additional elements allow for changing how the list is sorted, deleting students, and loading the screens that allow for entering new students or searching the student.

There are 2 main actions in response to user interaction. Firstly, the user can request that the active display show details on a single item, show the new student screen, or show the search screen. The actions involves interacting with a server object to update the student list, delete a student or perform a search.

The first type of actions are rather simple due to the fact that a DisplayMaster object perform user interface changes. Each of the methods in the displayPanel class that perform a user interface update actually delegates the call to a DisplayMaster object.

The second type of actions are more complicated. This is especially true when performing the search, due to the fact that call backs must be implemented.



The first server interaction code involves obtaining the sorted list of StudentI objects and then displaying them on screen. The user interface allows the user to sort students by student name as well as by student number but it use server to perform the physical sorting. The showStudentsByStudent() method listed below first obtains the sorted list of StudentI objects from the server, sticks them in a Hashtable object and then adds them to the List object for onscreen display. A Hashtable object is used for storage because it allows for easy reference of a single StudentI object when the user clicks one in the List object.

```
private void showStudentsByStudent() {
    StudentI[] students = _course.getAllStudentsByStNo();
    int iLength = students.length;
    _hshStudents = new Hashtable();
    _lstStudents.removeAll();
    for(int i=0; i<iLength; i++) {
        String sStudentName = students[i].sStudentName();
        _lstStudents.addItem(sStudentName);
        _hshStudents.put(sStudentName, students[i]);
    }
}
```

If user desires the students to be sorted by student number, the showStudentsByStNo () method is invoked. The method is identical to the showStudentsByStudent() method, save for the fact that it ask the server for the StudentI objects sorted by student name.

The second server interaction to focus on is the doDeleteSelected() method, which is invoked when the user clicks an item in the List object and then clicks the Delete Selected button. This method is passed a String object that's both the logical name displayed in the List object and the Hashtable key for the target studentI object. The method then uses the String object to obtain a reference to the target StudentI object, removes it from the server collection, and then removes it from the onscreen display.

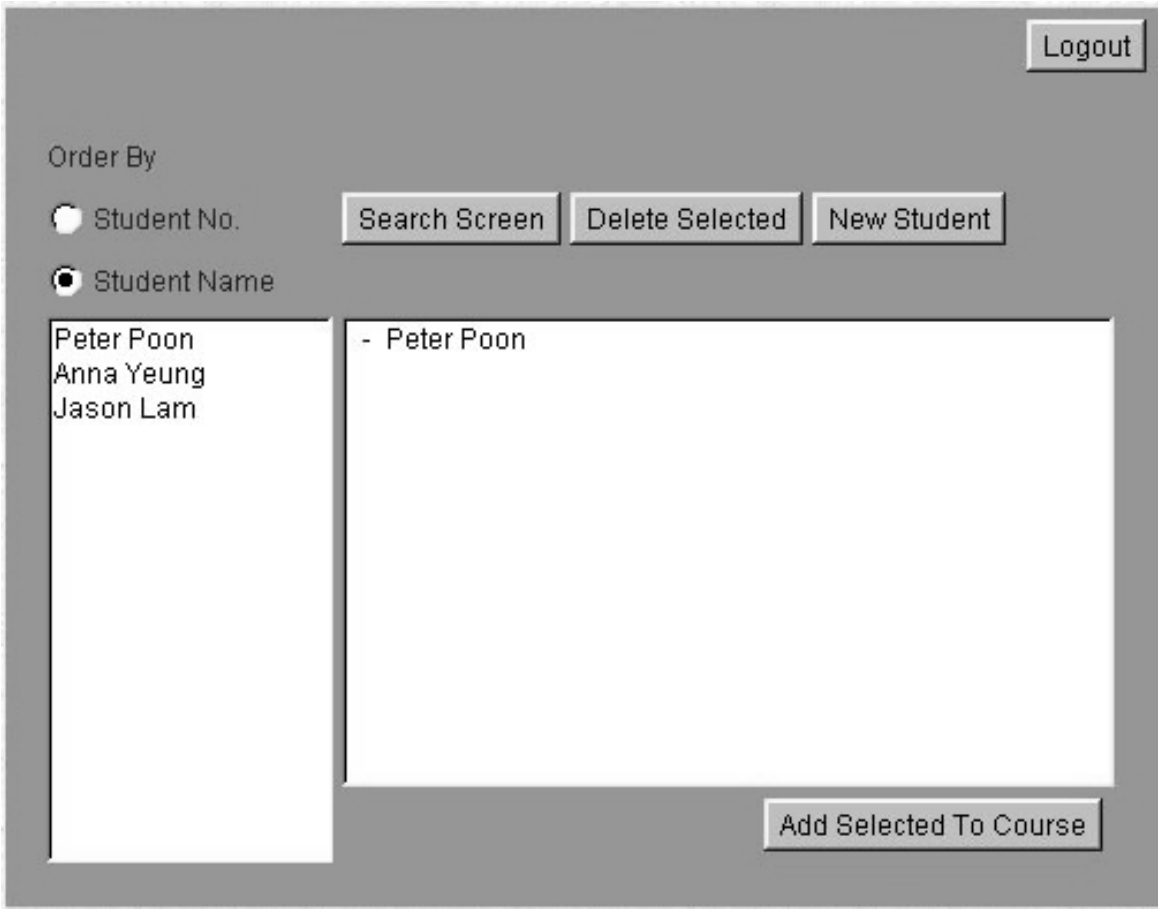
```
private void doDeleteSelected(String sName) {
    StudentI student = (StudentI)_hshStudents.get(sName);
    _course.deleteStudent(student);
    _lstStudents.remove(sName);
}
```

The next method doSearch() method and the ResultsListener inner class form the final server interactions performed by this class. This class implements the callback design pattern. Under the callback pattern, a client issues a query against a remote object but it is not returned the results of that query. Instead, the client passes a reference to another object that is then notified by the server when the results of the query are ready. The doSearch method is invoked when the user desires a search to e performed. The inner class ResultsListener represents the listener class in this situation. The doSearch() method is passed parameters that represent the user's search criteria. The method next packages up the search parameters in a StudentQueryS object, creates a new ResultsListener object, registers that object with the ORB, and finally invokes the searchCatalog() method on the server.

Once the searchCatalog() method is invoked, the code returns and simply waits for the server to finish searching for the students. When the server is finished searching, it invokes the studentFound() method in the ResultListner object and passes an object reference containing the results of the search. With the search complete, the studentFound() method requests that the displayMaster object display the result set.

```
public void doSearch(String sStudentName, String sStNo, float iCredit, Degree type) {
        // create the query object
        StudentQueryS query = new StudentQueryS(sStudentName, sStNo, iCredit,
type);
        // create the callback listener
        ResultsListener listener = new ResultsListener();
        // register the callback listener with the ORB
        _boa.obj_is_ready(listener);
        // perform the search
        _courseServer.searchCatalog(query, listener);
    }

        class ResultsListener extends _RequestorIImplBase {
        public ResultsListener() {
            super();
        }

        public void studentFound(StudentI[] students) {
            _displayMaster.showSearchResults(students);
        }
    }
```

ResultDisplay Class



This class is instantiated and displayed by a DisplayMaster object once a search is complete. The class displays a list of all students in the found set and gives the user the option to add any of those students to the course. In general, this class is simpler than previous class. The user interface presented by this class is simply a List object displaying all found students and a button object that adds the selected object to the course. When the button object is clicked, the actionPerformed() method is invoked. Then a reference to the target StudentI object is obtained, and it is added to the user's collection.

DisplayMaster Class

The DisplayMaster class is instantiated only once in the application and is charged with changing the main application display. The class extends java.awt.Panel and is added to the screen by the DisplayPanel class to the immediate right of the List object containing all available StudentI objects.

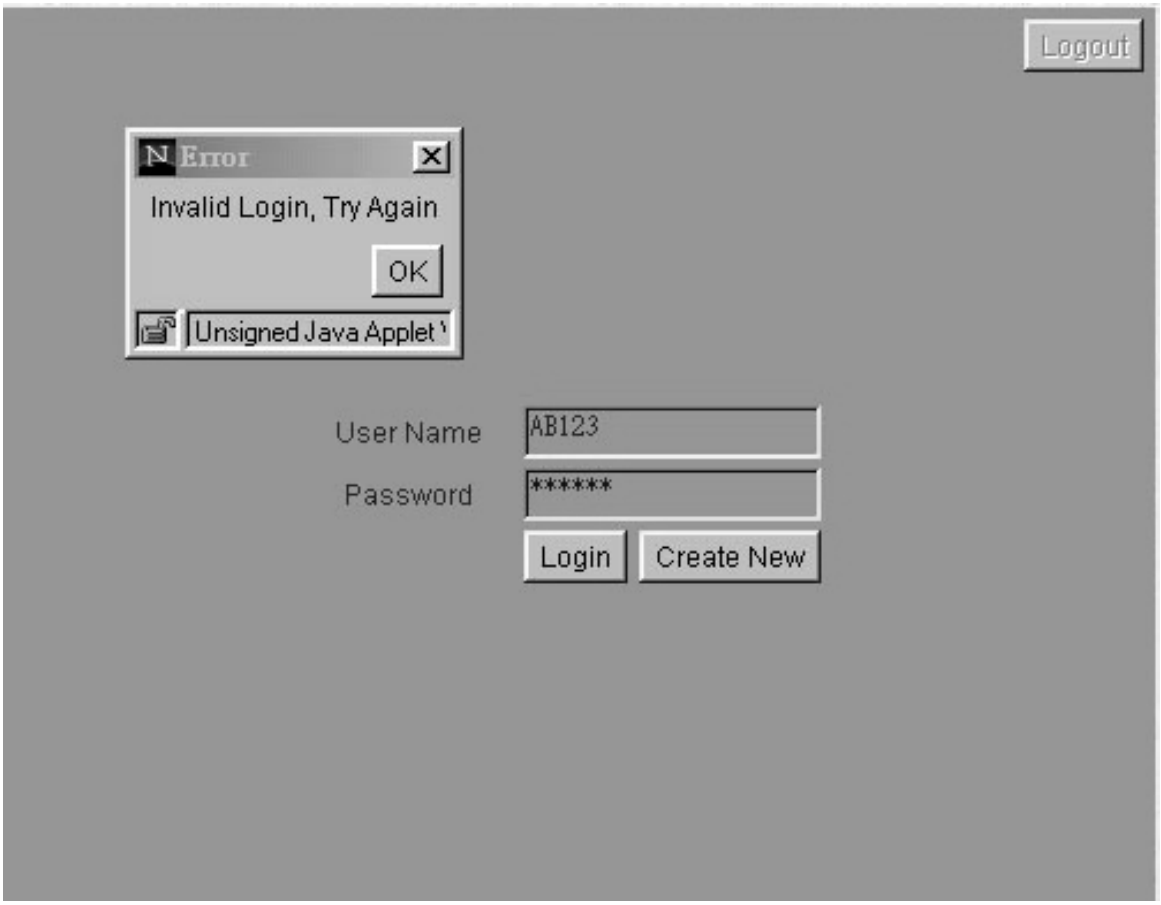
LoginPanel Class

This class present a user interface that allows a user to enter a login ID/password and then attempt to either log in or create a new account. The actual login

or create account attempt is not performed by the LoginPanel but instead happens in the CourseViewer class, which is passed the necessary information by the LoginPanel class.

ErrorDialog Class

This class is passed a parent Frame object and a String object that logically describes some error condition. In the constructor, a user interface is built that contains the error message and also a n OK button used to close the dialog box. This class is used to indicate that either an incorrect login ID/password combination has been specified during the login process or that the user has chosen a user ID that's already in existence



during the "create new account" process.

Running the Server and Client

Client as Applet

The client is designed to be an applet running in the browser. The suitable browser is Netscape Navigator installed with updated ORB. In order to house the Java applet, we need to write a HTML page. In the APPLET tags inside the HTML page, we need to add certain parameter to some special purpose:

```
<APPLET code=CourseViewer width=500 height=390>
<PARAM name=org.omg.CORBA.ORBClass value=com.visigenic.vbroker.orb.ORB>
</APPLET>
```

We should note that there is a parameter called org.omg.CORBA.ORBClass. This parameter is specially for Netscape Navigator 4.0 or greater that are currently bundling an older version of VisiBroker for Java (version 2.5). The older version is the default version used unless this parameter is set to the following:

Com.visigenic.vbroker.orb.ORB

Setting this forces the Applet to download the ORB classes that are known to the Web Server, which will be the latest version of Visibroker for Java. This setting, however, will drastically increase the download time.

Java Sandbox Problem

We have developed the applet and write the necessary web page to house the

applet. Are we ready to run the application? There is a Java sandbox some problem needed to be solved.

Java applets are running in the Java sandbox within the web browser. In an effort to protect the client host machine from potentially harmful or malicious applets, the sandbox security model provides restrictions on what the applet can do after it has been downloaded onto the browser. They are as follows:

1. Applet cannot access the local hard disk.
2. Applet cannot open network communication with any other host machine other than the codebase machine. One of the main advantages of distributed system is that processing can be performed by multiple objects on multiple physical hosts to provide scalability. The sandbox restriction eliminates his benefits.
3. Applet cannot use UDP broadcasts.
4. Applet cannot accept incoming communication. This prevent applet from being able to host any object implementation such as standard Callback objects.

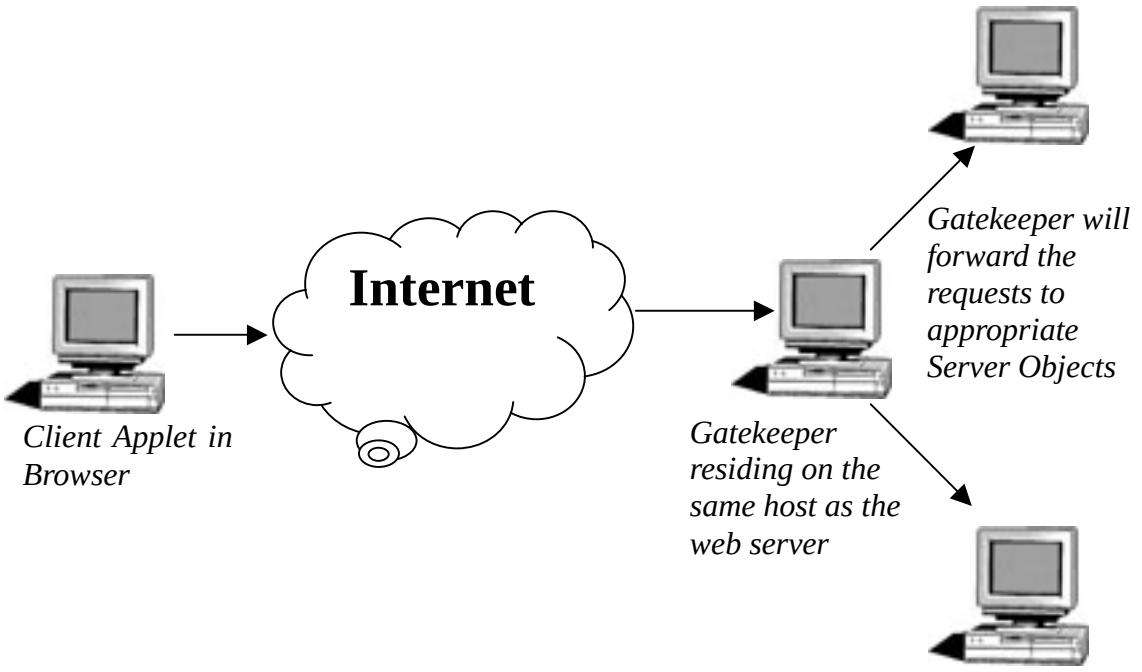
In order to overcome these restrictions, certain software is required. The Visibroker provide a “Gatekeeper” to make it appear to the developer that the security restrictions have been lifted, and thereby developing applets simpler.

Gatekeeper

The gatekeeper provides a number of capabilities. Firstly, it provides a

proxying functionality. We use the term proxy to describe the stub code that a client uses to communicate with a remote server. The proxying capability of the Gatekeeper extends this notion by effectively adding another level of indirection.

The applet in browser has a reference to an object running on a server which might be running on a machine other than the codebase. To the client applet, the object reference it is using looks like the real object in every respect. However, at TCP level, the applet is really sending a message to proxy object running on the Gatekeeper. The Gatekeeper forwards the message from the applet client to the actual server. The server's reply to the message is sent from the server to the Gatekeeper and then back to the applet client.



This extra level of indirection provided by the Gatekeeper is completely transparent to the developer. So, for example, if the server object reference is passed to some other process that is not an applet, then that application will communicate directly with the server and completely bypass the Gatekeeper. Furthermore, since this proxying is done by simply forwarding the IIOP packets from the client to the server, the proxying is vendor neutral. This means that the server is not limited to being a Visibroker server: It can be any IIOP capable server from any vendor.

The Gatekeeper's proxy forwarding capability can run in both directions, meaning that it can be used to expose callback objects created in the applet to other running CORBA processes. The Gatekeeper can send proxies from the LAN to the Internet and forwards packets from the server application to the client applet's callback object. Again, this proxying is transparent to the developer.

It can be assumed that if a user is running an applet, then clearly the user was able to download the applet in the first place via HTTP, the communication protocol used to access HTML documents. That is, the user must have HTTP connectivity back to the codebase to have obtained the HTML file containing the applet. If the applet was obtained from the codebase via HTTP, it should be able to communicate with codebase via HTTP. Of course, CORBA does not use HTTP for communicate. It use IIOP instead. To work around this problem, the Gatekeeper implements not only the IIOP protocol, but also implements the HTTP protocol. For simplicity of administration, the Gatekeeper implements both protocols on the same port on the server. To do HTTP tunneling, the client simply wraps its IIOP requests inside an HTTP "get" request by

encoding the IIOP requests as a URL.

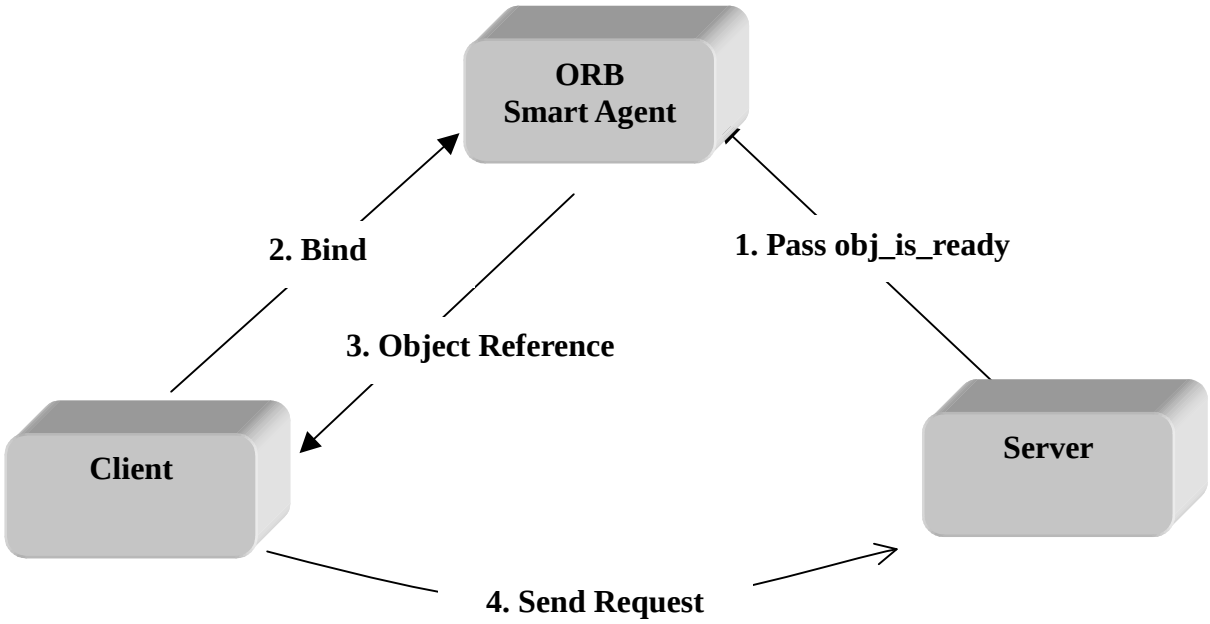
ORB Smart Agent

As we have discussed, the gatekeeper help the client applet to forward the request to the remote objects by the HTTP tunneling. However, we need to know the location of the remote object and how to connect to the remote object. This problem of how to find initial entry into a self-navigable system is often referred as bootstrapping. This is where the CORBA specification does not make any requirements. The CORBA 2.0 specification itself doesn't mandate any particular type of convention for locating remote objects. The specification simply discusses how an object implementation makes itself available to start receiving invocations and how it creates a unique reference for itself, the Interoperable Object Reference (IOR). The IOR of an object implementation is the unique identifier of an object implementation, providing all the information necessary for another CORBA process to locate and communicate with it.

The Visibroker developers created their own proprietary Directory Service called Smart Agent. The Smart Agent provides dynamic, distributed directory service. A overview of how the smart agent works is as follows:

1. A Smart Agent started on the network.
2. A Server Object is started and registers its location information with the Smart Agent.
3. A Client is started and calls the Smart Agent requesting the location information of the Server Object.

4. The ORB instantiated a local proxy object, or stub, which has the necessary IOR details of the real Server Object. The Client is returned to a local reference to the stub that handles all the data marshaling and network communication to he actual Server Object implementation.



The Smart Agent can help to provide object implementation fault tolerance. We can provide object implementation failt tolerance for objects by simply starting instances of those objects on multiple hosts. The ORB will detect the loss of the connection between the client application and the object implementation and the ORB will automatically attempt to establish a connection with another instance of the object implementation. The client can continue invoking methods on the object without being concened that a new instance of the object is being used.

Server Deployment - Object Activation Daemon

In the building of CORBA application, a simple development environment is desirable. All work typically occurs within a single subnet, and the developer usually maintains manual control over the starting and stopping of servers. All these assumptions will not hold when the application is deployed. It is not manageable for a deployed application to require all servers to be manually started. In large systems containing thousands of object implementations, manually startup of objects can prove to be very inefficient. In addition, the distributed application may be designed to have components spanning multiple subnets.

The Visibroker provides an Object Activation Daemon for Java, oadj, which is designed to provide automatic activation for Server Objects. The oadj works in conjunction with the CORBA Implementation Repository to start up object implementation on demand. The implementation Repository is similar to the Interface Repository. The Interface Repository is designed specifically to store information on Visibroker interfaces, whereas the Implementation Repository is a database of the actual implementations themselves. The oadj relies on the information stored within the Implementation Repository in order to start Server Objects.

In simple client to server object communication, the model is as follows:

1. Start the Server Object. It registers with some Directory Service. This could be Smart Agents, COS Naming Service or simply passing Interoperable Object

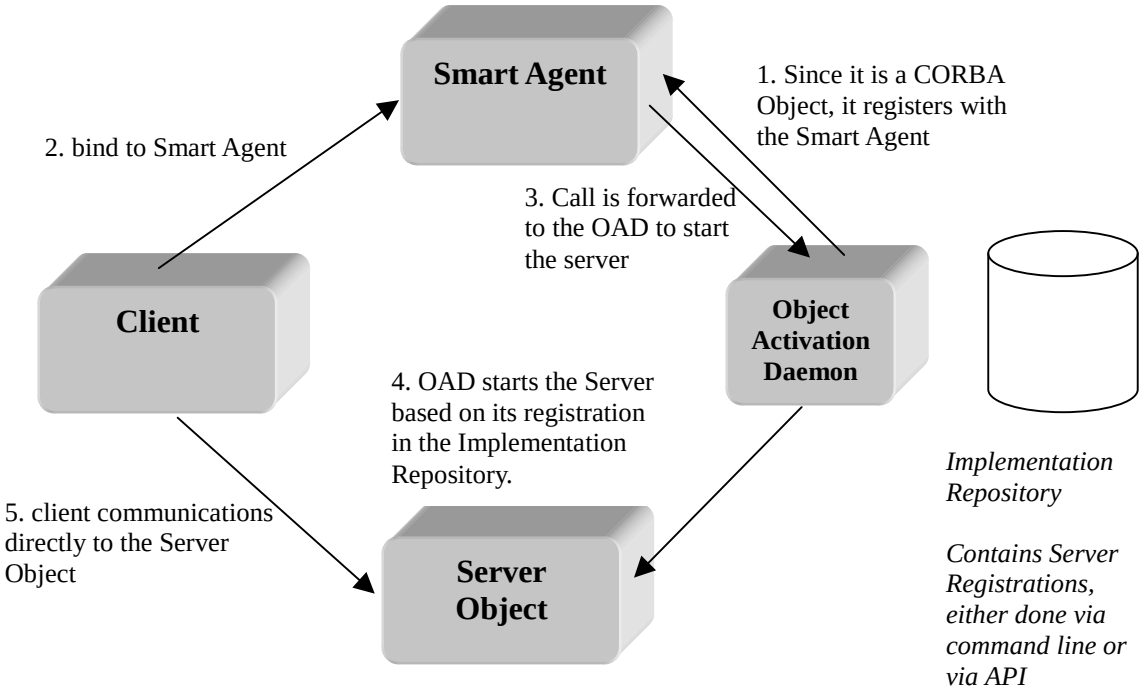
References (IOR) directly.

2. Start the Client. It uses one of the location mechanisms to retrieve the Server object IOR.
3. The Client communicates directly to the Server Object.

When we use the Object Active Daemon, the model is like this:

1. Start Smart Agent
2. Start the oadj. Because it is a CORBA object, just like any other Server Object, it needs to register with the Smart Agent.
3. Register the Server Object with oadj. This can be done either programmatically through the IDL interface to the oadj, or through using a command-line utility.
4. Start the Client. It attempts to locate the Server Object via the Smart Agent. Because the Server Object itself isn't running, there is no registration for it in the Smart Agent's memory table. There is however, an entry for the oadj. Thus, when the request comes to the Smart Agent, the Smart Agent returns the reference of the OAD to the client.

The client makes an invocation to the Server Object. However, this call actually goes to the oadj. The oadj then starts the Server Object and forwards the call to the newly spawned Server Object.



Activation Policies for the OAD

The OAD activities the Server Objects that are registered with it, based on its activation policy. The activation policy also indicates the lifespan and behaviour of the Server Objects as well.

The CORBA specification defines 4 activation policies:

1. Persistent Server
2. Shared Server
3. Unshared Server
4. Server-per-method

The Persistent Server policy is actually common in use. Within the Visibroker

context, any Server Object that is started manually outside of the OAD is created with the Persistent activation policy. This is true for Server Objects that are created as globally scoped objects or local transient objects. It isn't possible to have the OAD start a Server Object with the Persistent policy.

The Shared Server policy means that a single Server Object implementation is spawned and is shared by all Clients that attempt to bind() to it.

The Unshared Server policy means that every client that attempts to bind to a Server Object causes the OAD to spawn a dedicated instance specifically for that Client. The Client maintains a connection to its own instance of the Server Object and is the only Client connected to that particular instance. Thus, the Server Object instance is around as long as the Client maintains a connection. After the Client drops the connection, the Server Object instance is marked for garbage collection.

The Server-per-method policy means that a Client receives a new instance of the Server Object for each method invocation. The Server Object instance exists for the duration of the method and then is marked for garbage collection after the method has completed. Each subsequent method invocation results in a new Server Object instance being created.

Server Deployment – Naming Service

In a distributed environment one of the more complicated tasks one must deal with is locating objects needed at a given point in time. Large distributed environments may exist such that hundreds or thousands of different objects all publish some level of functionality that other objects will want to take advantage of. Of course, it is possible to give each object s different logical name, but this is not easy as it may sound. Just as it is difficult to name humans in an easy to understand manner, it is also difficult to name distributed objects in an easy to understand manner. In an attempt to ease the object naming problem, the OMG has released a specification for a context-sensitive naming scheme called the Naming Service. The Naming service allows for context-sensitive names to be associated with objects, and for those objects to be referenced under their names. The Naming Service has been introduced in the previous chapter on CORBA Object Services (COS). We will go on to discuss the Naming Service in more detail and have an implementation using Naming Service.

Background

Basically, a Naming Service provides a mechanism for associating objects with a logical name within a searchable structure. A good place to start our model of a Naming Service is with a familiar structure, a file system. A file system contains files as leaves hanging off of the nodes. The files are referenced by the file's name. Therefore, the object is referenced by a property of that object. In a Naming Service, the name of the object is logical concept, and is not a property of the object. When we place a file into a directory, the file system places that file into the file system. For instance, the act

of creating a file in a file system both instantiate that file and binds it to a name. However, with the Naming Service, the user has more control over this act of placing names into the tree. This act of associating a name with an object is called binding. When we place a file in a file system, the binding is implicit. With a Naming Service, the binding operation is explicit.

Start the Naming Service

The Naming Service specification describes the entire namespace in terms of NamingContext objects. These contexts can be connected to each other and can contain references to actual object instances for which clients will ask. In the Visibroker Naming Service, these contexts are all contained within the CosNaming.Factory and CosNaming.ExtFactory servers. When the factory server is executed, it always creates a singleton persistent object implementing the CosNaming::NamingContextFactory interface. All NamingContext objects created within a particular factory server are associated with that server's single NamingContextFactory.

When the basic CosNaming.Factory is started, only an implementation of the NamingContextfactory interface is created, without any Namingcontext objects. A client must connect to the NamingContextFactory and ask it to create a first context. The reference to this context would have to be stored in some repository for a subsequent client to be able to contact it. Typically, an application requires a rooted hierarchy of naming contexts and prefers an easy way to get to the root of this hierarchy. To this end, Visibroker's Naming Service provides a second factory, the

ExtendedNamingContextFactory. When the CosNaming.ExtFactory server is executed, an instance of this extended interface is created, as well as a single NamingContext that will be that factory's root context.

When a factory server is started, it is given a factory name and the name of a repository file to be used for logging. The server's singleton factory is given the factory name as its object name. This is the name that will be registered with the Smart Agent. All of the contexts associated with this factory will also have this factory name embedded in each of their object names. The actual name of each of the contexts created by a given factory is of the form <factory name>/<context number>, where the factory gives each created context a unique number. For example, the first context created by a factory named "Course" would be "course/1", the second would be "Course/2".

Each time a context is created, destroyed, or modified in any way, an entry is placed in the server's log file. Each server must be started with its own log. If a factory server is shut down and restarted with the same log it had previously created, the exact state of the server will be restored, including the contents of all NamingContexts.

Publishing Object References

Once a NamingContext is available, object references can be bound to logical names within it. The task of registering top-level objects into the namespaces can be owned by the objects themselves or by a separate administrative entity. We will use the server mainline code to bind its newly created instance to a logical name within root

NamingContext in our implementation of server.

Implementation of Application with Naming Service

We will build a distributed student management application with the use of COS Naming Service. The server will store a number of student. The client application can browse the server's list of student or add new student. Student can be selected and add to the client side list.

The IDL of the application is simple:

```
interface StudentI {
    attribute string sStudentName;
    attribute string sType;
    attribute string sAdditionalNotes;
    attribute long    credit;
    attribute long    points;
    attribute string sPointSource;
};

typedef sequence<string> stringSequence;

interface StudentServerI {
    void addStudent(in stringSequence categories, in StudentI student);
```

};

Only one method is available at the StudentServer. It allows the addition of new student. New student added can then be categorised under different naming space.

StudentServer Class

In the StudentServer class, the process of binding the object to the name graph takes place in the bindObject() method. All binds were performed under the assumption that the NameComponent object being bound may or may not be in use. Every call is then proceeded with an invocation of the resolve() method. If the resolve() method does not throw an exception, we know that the NameComponent object is already part of the name graph. If an exception is thrown, we know that the NameComponent object is not present in the name graph and can proceed to add it. We then call the bind_new_context method to bin each NameComponent object to a new context. Lastly, we call bind() method to bind the last NameComponent object to the NameServiceObject object.

```
public void addStudent(String categories[], StudentI student) {
        NamingContext root = _root;

        int iLength = categories.length;
        iLength--;

        NameComponent[] componentHolder = new NameComponent[1];
```

```
        for(int i=0; i<iLength; i++) {
            componentHolder[0] = new NameComponent(categories[i], "");
            // see if the context is already bound
            try{                                                    root          =
NamingContextHelper.narrow(root.resolve(componentHolder)); }
            catch(    Exception e ) {
                    // bind the new context
                    try{       root = root.bind_new_context(componentHolder); }
                                                    catch(       Exception    innerE    )
                        { System.out.println("inner: "+innerE); }
            } // catch
        } // for

        // create a copy of the original StudentI object
        StudentI newStudent = new Student();
        newStudent.sStudentName(student.sStudentName());
        newStudent.sType(student.sType());
        newStudent.sAdditionalNotes(student.sAdditionalNotes());
        newStudent.credit(student.credit());
        newStudent.points(student.points());
        newStudent.sPointSource(student.sPointSource());

        // activate the new object
        _boa.obj_is_ready(newStudent);
```

```
        componentHolder[0] = new NameComponent(categories[iLength], "");
        // bind the object
        try{        root.bind(componentHolder, newStudent); }
        catch(    Exception e ) { System.out.println("e: "+e); }
    }
```

StudentClient Class

In the StudentClient class, an important method is obtainNameGraph method. In this method, the NamingContext.list() method is executed recursively, giving a picture of all NamingContext and StudentI object represented by the root node. In addition to traversing the name graph, the obtainNameGraph() method places all found information in a hierarchically formatted list object.

There are two other GUI classes to help the display of student and help to add new students.

Conclusion

The Common Object Request Broker Architecture (CORBA) is architecture of distributed object computing. Its feature of platform and language independence provides a strong base towards its use in both old and new systems. Commercial products such as Iona Orbix or Inprise Visibroker that meet the CORBA specification are available for system development. The CORBA ORBs are also widely available in different platforms such as UNIX, Windows and Mac, etc. With the help of IIOP, objects implemented in a wide variant of operating system are still be able to be communicate between different ORBs.

Obviously, we have named many advantages of CORBA here and also in the previous chapters. However, there should always be something more important than the others. We would discuss them below and also their importance in a web learning environment with CORBA. We will then go on to see some downsides of CORBA and particularly in the web learning environment.

Upside of CORBA

Language and platform independence

There are many different rival technologies in the world that may be able to replace CORBA. What key feature can we make CORBA stand out from the others? Language and platform independence is obviously something important. Other technologies such as DCOM or RMI are either platform or language independent. They

need to stick to their platforms or languages. When we face the real world development environment, we are likely to encounter different operating system or system implemented in different language. Language independence can help to increase the productivity of system developer. New programs can be written in any language familiar with the programmers provided that an IDL compiler available for this language. Today, we already have a lot of different IDL to other language mappings and also the compilers available to facilitate the CORBA development.

In our web learning environment, the obvious choice of language is Java as it is directly support to run in different browser environments. However, people may wonder if Java is really good. Someone may wonder if it is a good partner with CORBA too. Even though Java may become obsolete some day in the future and we have another popular language, new CORBA language mapping will come out. Hence, CORBA can still help the system development in future.

On the other hand, we have platform independence works to help to facilitate the role of CORBA as a middleware. In the past, the maintenance of the old system is almost always time-consuming and costly. If we want to extend the functionality of legacy system, it may be likely that we may end up a costly system that is even more difficult to maintain. Building a new system may not always be a viable option. With CORBA, we can wrap the old legacy system with object wrappers. The legacy system may be treated as one or more objects to be able to access via the CORBA. Thus, we save a lot on the system maintenance.

In web learning environment, it may be likely that we need to extract certain data from an old system. It may be a student record system. We can use CORBA to help us to connect to this system to extract valuable information easily.

Distributed Object Technologies

CORBA is a distributed object technologies. It is an integration of object and distributed system. With object based design, it is more easy to emulate the real world objects with software objects. System developers can improve productivity with an object oriented design. Different design patterns are available to let users to learn from the experience of other system designers. As computer system change from centralised one to simple networked system or client server design, the design of distributed system become more important. However, the heterogeneous platforms in a distributed system is almost always a problem to solve. With CORBA as middleware, distributed system with heterogeneous platforms no longer too difficult to build. When we bring object design and distributed system together, we have distributed object computing. In the real world system, objects would be located in different hosts. The CORBA provided services to give you location transparency and yet you can interconnect them to form a working system. The distributed objects can also improve load balancing particularly when object migration is possible in the distributed system.

The web itself is already a distributed environment. The learning environment built around the web is always intuitively a distributed system. We can easily make use of the different services provided by CORBA to build a good distributed system. These services include Transactions, Naming, Security and Events.

CORBA is Open

CORBA is an open technology. It came from the collaborative efforts of different people and companies. The consortium behind CORBA consists of virtually every major vendors in the industry. No single vendor control the technology and it is well receive by the software industries as standard. Having all the major players behind a specification ensures that it never becomes too slanted toward the needs of any single vendor. Because of the openness of the CORBA, we always have the choices to choose a product for development. We can build a system with the use of Visibroker for Java or we can build it with OrbixWeb. A hybrid system using both products are not impossible too.

As more vendors or researchers join to work in different business area, CORBA domains will become rich enough to ease the software development processes in these areas. We have CORBA Telecom, CORBA Finance and CORBA Med today. Although we do not have something called “CORBA Learning” today, it may come out at one day and lay a solid foundation to the development in web learning environment.

CORBA’s Problem

Complexity of Development

Software development in CORBA requires an understanding of many distinct but interrelated topics. It is a steep learning curves from the requisite knowledge of the interplay between ORBs, object adapters, object naming and object activation to the

concepts dealing with server robustness such as error handling, garbage collection, loading balancing, threading, interceptors, transaction management and object persistence. Throw in application design topics such as server callbacks, firewalls, security issues and various useful CORBAServices just to make things interesting. While different ORB implementations have much in common conceptually, each ORB vendor imposes its own perspective of these topics on the developer. It can be absolutely maddening when the developer is also in the process of learning the problem domain, OO, CORBA, IDL or Java.

Maximizing developer productivity is the mantra of corporate IT departments, and in order to make CORBA development cost-effective, tools are needed. Fully functional development tools that simplify and abstract most of these complexities, that provide seamless support for component technologies like JavaBeans, and that integrate application deployment and management facilities are not currently part of the CORBA landscape. The things we need are CORBA development environments that provide robust support for easily deploying and administrating distributed applications. Developing CORBA applications can be a difficult, tedious task, which requires skills and capabilities beyond those typically found in mainstream application developers. A suitable tools with integrated ORB, CORBAservices, graphical object management and monitoring tools is required.

Cost of Development

Depending on the scale of the applications intended to develop, implementation of application in CORBA architecture would make the development cost much higher

because of complexity and lack of experienced developers. It would be unwise to develop simple application in the complex CORBA environment if there is other means to do so. The cost of the CORBA software or the ORB is sometimes a prohibitive cost to the development of application in CORBA. The significant investment in the CORBA ORB and software will cause some organization to exclude CORBA when making decision. Rival technology like DCOM ships for free with all version of Windows NT and 98. The lack of an initial investment may draw organization to use DCOM.

Stability

Stability of the application is sometimes important to an organization. In the real world enterprise client-server applications, users expect the applications to be robust with a low failure rate. Because of the ever changing nature of new technologies like CORBA and Java, the stability of CORBA development tools and also the final application product may be in question.

All Come Together

When we consider all these things together, CORBA is still an architecture useful to the development of a web-based learning environment. Its initial development cost may be large when compare with other technologies. However, most web based learning environment is expected to handle large number of students. It is easy for the CORBA applications to scale up to handle more users. It is a promising technologies that help us to join different systems together into play. Since we don’t know what other emerging technologies may become useful in web based learning in future, CORBA make

it easier to incorporate outside system in our web environment. It is only a matter of time that CORBA will be a mature technologies to be fully in use in web based learning environment.

References

1. Object Management Group. The Common Object Request Broker: Architecture and Specification. Object Management Group , Framingham, M.A., Revision 2.2, July 1998
2. Object Management Group. Common Facilities Architecture. Object Management Group , Framingham, M.A., Revision 4.0, November 1995
3. Object Management Group. CORBAservices: Common Object Services Specification. Object Management Group , Framingham, M.A., December 1998
4. David Curtis, Christopher Stone and Mike Bradley. IIOP: OMG's Internet Inter-ORB Protocol: A Brief Description. Object Management Group , Framingham, M.A., May 1997
5. David Curtis, Java, RMI and CORBA. Object Management Group , Framingham, M.A., May 1997
6. Kate Keahey. A Brief Tutorial on CORBA, http://www.cs.indiana.edu/hyplan/kksiazek/tuto.html
7. Zhonghua Yang & Keith Duddy. CORBA: A Platform for Distributed Object Computing
8. Sean Baker. CORBA Distributed Objects:Using Orbix. Addison, Wesley, Longman, November 1997
9. Andreas Vogel and Keith Duddy. Java Programming With CORBA. John Wiley & Sons, February 1998
10. Doug Pedrick, Jonathan Weedon, Jon Goldberg and Erik Bleifield. Programming With Visibroker : A Developer's Guide to Visibroker for Java. John Wiley & Sons, March 1998
11. Michael McCaffery and Bill Scott, The Official Visibroker for Java Handbook, Sams,

1999

12. Bill McCarty and Luke Cassady-Dorion, Java Distributed Objects, Sams, December 1998

13. Robert Orfali, Dan Harkey and Jeri Edwards, CORBA, Java, and the Object Web, Bytes, October 1997

14. Lewis, Geofrey, Steven Barber, and Ellen Siegel. Programming with Java IDL: Developing Web Applications with Java and CORBA, John Wiley & Sons, 1998

15. Mowbray, Thomas J. and Raphael C. Malveau, CORBA Deign Patterns, John wiley & Sons, 1997

16. Orfali, Robert and Dan Harkey and Jeri Edward, Instant CORBA, John Wiley & Sons, 1997