A Distributed, Component-based Solution for Scientific Information Management

Lalit Kumar Jain

A research paper submitted in partial fulfillment of the requirements for the degree of Master of Science

Major Advisor: Dr. Timothy A. Budd

Department of Computer Science
Oregon State University
Corvallis, OR - 97331 - 3202

July 1, 1998

Abstract

This paper describes the design and performance of a distributed, multi-tier architecture for scientific information management and data exploration. A novel aspect of this framework is its integration of Java IDL, the CORBA distributed object computing middleware with JavaBeans, the Java Component model to provide a flexible, interactive framework for distributed, high-performance scientific data visualization. CORBA server objects running in a distributed, collaborative environment provide data acquisition and perform data-intensive computations. Clients as Java Bean components use these server objects for data retrieval and provide an interactive environment for visualization. The server objects use JDBC, the Java application programming interface to databases, to retrieve data from the distributed data stores. We discuss the system framework and its components and describe an example application and its performance.

Table of Contents

Chapter 1
Introduction

This project is a contribution to the ongoing work at the college of oceanic and atmospheric sciences which is part of NASA's Earth Observing System project. The main goal of the work is to design and develop an information management and data exploration system to provide easy and meaningful access to the ever-increasing quantity of earth and space science data. Such a system would enable researchers to easily and efficiently access the required data sets and undertake complex analysis to uncover and extract relationships hidden in those large data sets. Consider that many of these data sets need to be compared with other data sets as part of the analysis process and also the data sets of interest may reside on different machines in different formats. Simply putting these enormous data sets on-line neither addresses the problem of locating and retrieving required data nor does it facilitate effective data analysis.

Earlier efforts at creating such a system coupled with changes in technology and application environments have resulted in an evolution of the current design. The client/server model, though, has remained the dominant computing architecture. The feature that distinguishes client/server from other technology architectures is that it contains cooperative processing capabilities, which physically split the processing performed by the client from that performed by the server while presenting a single logical picture to the user. Two-tier client/server approaches with client applications directly accessing data stores and retrieving requisite data resulted in monolithic, inefficient, non-scalable systems. Such systems were also unsuitable for data access over the internet as they suffered from the fat-client problems.

A two-tier design paved the way to a more scalable three-tier approach. Such a design partitions the system logic into an application tier and a presentation tier. The application tier encapsulates the data access and processing logic and allows the presentation tier to concentrate solely on data exploration and visualization. It has several advantages. Separation of the presentation logic from the data access logic enables the data exploration logic to be independent of the data access logic. The middle tier is responsible for locating and retrieving the required data and making it available to the presentation tier in a suitable format. Modularly designed middle-tier server modules can be reused by several applications. Middle-tier functionality servers are highly portable and can be dynamically allocated and shifted as the needs change. It also results in thin-clients which are suitable for running on the internet.

In earlier client/server systems, the mode of communication between the application tier and the presentation tier has been the Remote Procedure Call mechanism. Successful use of the object-oriented programming paradigm has led to an object-oriented communication mechanism and the concept of distributed objects. Objects break up the server and client sides of an application into smart components that work together and operate across networks. CORBA and RMI are two popular distributed object standards. RMI (Remote Method Invocation) is an API standard for building

distributed Java systems. RMI provides a simple framework for Java-based object communication. It is very straightforward and easy to use for developers familiar with Java. CORBA (Common Object Request Broker Architecture) is an architecture standard for building heterogeneous distributed systems. It is a language and architecture neutral standard allowing objects written in different languages and running on heterogeneous platforms to communicate with each other. It has several benefits which makes it the best distributed computing middleware. RMI is specifically not capable of handling very massively distributed systems. It is at this time missing some of the key services that CORBA supports. These services such as transaction management are a requirement of large distributed object solutions.

This report presents a multi-tier client/server software architecture that has been designed for scientific application domain in general. It describes the development of a distributed application for information management and data exploration for the oceanography domain in particular.

From the software engineering point of view, this project represents an effort at understanding the principles of object-oriented analysis and design and the process of building object-oriented frameworks. A concerted effort has been made to make maximum use of design patterns for addressing design issues in order to achieve high levels of design and code reuse and also simplify understanding of the system.

Chapter 2
Distributed, Component-based Software Development

In the recent past there have been several revolutions in the software development industry, all focussed towards the evolution of a standards-based software engineering methodology which would enable development of software application using standard software parts much in the same way as products in other engineering fields are developed. The shifts from monolithic to client/servers systems and to distributed and multi-tier systems promised to simplify the development and maintenance of complex applications by partitioning centralized systems into components that could be more easily developed and maintained. The object-oriented paradigm revolutionized the way software was designed and implemented. It enabled larger applications to be modelled in term of smaller, interacting objects each of which encapsulates its own state and behavior and communicates with other objects through message passing. It was a step towards a technology which promised reusability and maintainability of software. The application of object-oriented paradigm to distributed applications was a natural step. The advent of component-based development model is the next major leap which helps to identify software in terms of components which have the ability to be assembled to form applications. It leverages the object-oriented paradigm in terms of reusability. The next step then is towards the integration of component-based development scheme with the distributed object computing paradigm to define a component-based, distributed-object computing model.

2.1 Distributed Object Computing with CORBA

The object paradigm has successfully been applied to the design and implementation of application frameworks, graphical user interfaces, object-oriented databases, etc. Object-oriented programming of distributed applications is the next logical step. Ideally, accessing the services of a remote object should be as simple as invoking a method on that object.

Distributed Object Computing(DOC) is a variation of the client-server model. In DOC, objects encapsulate an internal state and make it available through a well-defined interface. Client applications may import an interface, bind to a remote instance of it, and issue remote object invocations.

The OMG CORBA standard permits the design and development of object-oriented distributed systems by providing an infrastructure that allows objects to communicate independent of any programming language and techniques used to implement the objects.

The Object Management Group

The Object Management Group (OMG) is the world's largest software consortium whose mission is to promote the theory and practice of object technology for the development of distributed computing systems. A key goal of the OMG is to create a standardized object-oriented architectural framework for distributed applications based on specifications that enable and support distributed objects.

Object Management Architecture

The Object Management Architecture (OMA) reference model is an architectural framework that identifies the key components of a distributed object system. The following are the various components of the OMA reference model

1. Object Request Broker

The Object Request Broker (ORB) can be thought of as the foundation of the OMA. It handles all communications between objects within a distributed object system. The ORB accepts requests from clients, locates and activates the target objects, and forwards the requests. All other system components depend on the base set of services provided by the ORB. It supplies the infrastructure that allows other components to work together to provide an integrated distributed object system.

2. CORBAservices

The ability to communicate, while important, is only a part of the overall solution. In order for objects to work effectively together, additional services need to be provided. Objects need to know where to find each other, whom to trust, and how to manage their life cycles. The OMG has defined a set of standard object services, known as CORBAservices, to address these issues. Services such as Naming, Security, Persistence, and Transactions are essential to developing robust client/server applications. By providing such services and defining consistent interfaces to them, application developers can quickly make use of components that distributed object application need.

3. CORBAfacilities

CORBAservices are system-level components; CORBAfacilities are application-level components. They cover such services as printing, document management, etc., all of which can be used by applications rather than forcing each application developer to reconstruct them using low-level system components.

4. Domain Interfaces

Domain Interfaces are specifications for objects or system components that are useful to specific vertical market segments. They provide very specific functionality useful to users in a certain field. Examples include components designed for markets such as finance, medicine, and manufacturing.

5. Application Interfaces

Even though application interfaces are components of the OMA, they are not explicitly defined by the OMG. They are intended to represent the objects written by various end users to meet their needs. They make use of all the other OMA components, facilitating reuse and rapid development.

The CORBA Architecture

The CORBA specification defines several key pieces that together make up the foundation of the entire OMA infrastructure.

Interface Definition Language

The Interface Definition Language (IDL) is the most significant element of CORBA. To be able to

create distributed objects, we must first be able to describe them, and IDL provides a formal mechanism for specifying an object's interface in a way that is independent of its implementation. This is one of the key reasons for the flexibility of the CORBA architecture. By separating the object's interface from its implementation, portability and interoperability can be ensured.

Object Request Broker

The ORB core consists of the underlying communications mechanisms and object representation. The ORB provides two methods of communication: static and dynamic.

Static method invocation allows clients to call object methods using client-side stubs specific to that object type. The stubs are generated from the object's IDL specifications by an IDL compiler and are linked with the client application at compile time. They provide the client with an interface to the object's method while hiding all the communication details.

Dynamic invocation allows requests to be constructed on the fly and sent to a target object. This is useful when the type of the target object will vary at runtime. The client does not know the exact type of the target, but using the object reference it can query the interface repository to determine the methods supported and their associated parameters. A request can then be dynamically constructed and sent to the object. This allows extremely flexible and dynamic applications to be created that make use of the CORBA communication mechanism.

Interface Repository and Implementation Repository

For the ORB to be able to talk to different objects, it needs to be able to determine the types of those objects. This is where the interface repository comes in. All objects in a CORBA system store their type information in a database known as the interface repository. The ORB uses this information to forward requests, to check that parameters are correct, and to support dynamic invocation.

In addition to the interface repository, CORBA provides an implementation repository. This is where the ORB stores information about each object's implementation. The information in this database depends heavily on the particular implementation and can include such things as the location of the server executable, the machine on which to run the server process, and startup arguments.

Invocation Scenario

We will now look at what happens when a request is made. Starting at the client we will follow a request as it goes through the ORB and up to the target object.

Client

Before a client application can invoke a method on an object, it must first obtain a handle to that object. In CORBA, these handles are called object references. An object reference usually contains the information that allows the ORB to locate and forward a request to that object. What exactly that information is depends on the ORB implementation. The ORB is solely responsible for generating and interpreting object references.

Once in possession of an object reference, the client must choose an invocation mechanism.

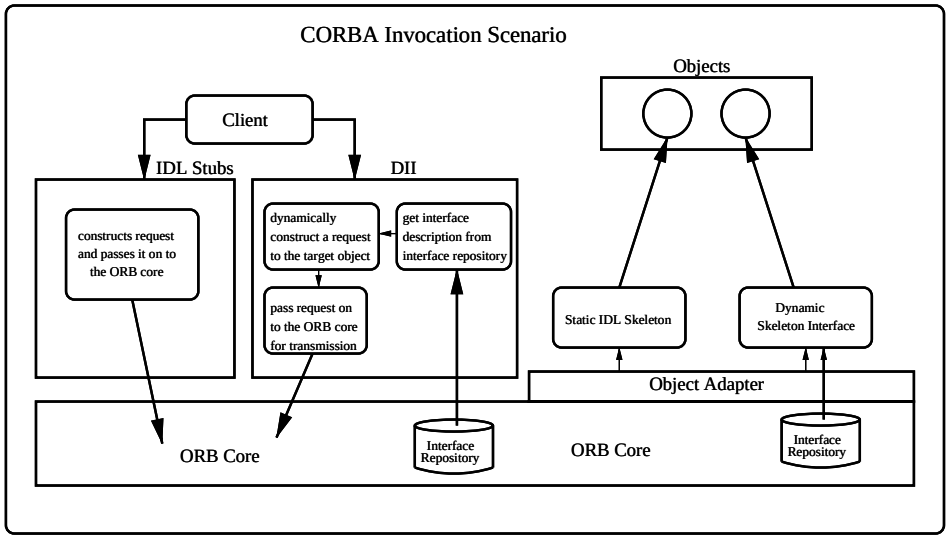
Depending on the situation and requirements, the client application will use either static or dynamic invocation. Regardless of which client-side mechanism is chosen, the request will be the same at the server.

If static invocation is used, the request is made by calling the desired method of the client stub object. The stub will convert the method parameters into an interoperable format and package them into a request structure to be sent to the server. This process is known as marshalling. After marshalling, the request is passed to the ORB core to be sent to the target object.

With dynamic invocation, the situation is slightly different. Rather than have a stub object perform the marshalling transparently, the application programmer constructs the request which holds everything pertinent to an invocation normally using functions provided as part of the ORB client-side interfaces. Naturally, the programmer needs to know the object type and parameters to each method to do this. To get this information, the programmer can query the interface repository using a well-defined interface.

After being issued by the client, the request is passed to the ORB. Once the ORB has the request, it determines the target object and how to contact it using a combination of the object reference and the interface and implementation repositories. The request is then transmitted to the target machine using an underlying transport such as TCP/IP.

The CORBA Invocation Scenario



Server

On the server side, the ORB must perform a number of functions before the request reaches the server object for processing. Usually the server-side ORB components receive the request over a network connection. The ORB needs to unmarshall the header data from the transmission format and determine which of the objects on that machine is the target. Having located the object, the ORB must prepare the object to receive the request. This could mean a number of things ranging from starting the server process that contains the object to retrieving the object from persistent storage, such as a database or a file.

Next, the ORB passes the request up to the object. This is done through an upcall to the object implementation from the ORB layer. The ORB core, however, does not talk directly to the object. Communication between the ORB and an object is handled by an object adapter. The object adapter provides an interface that allows a common ORB core to support a wide variety of different object implementations. For example, an object could be implemented using a library, a server process that contains the object, or a different process for each object method. The object adapter insulates the ORB core from these differences and allows those implementations access to the ORB core through the most appropriate mechanism. This flexibility is one of the primary strengths of the CORBA model.

Two up-call interfaces to the object lie above the object adapter: the static IDL skeleton interface and the dynamic skeleton interface. They are the server-side analogs of the static and dynamic stubs and provide essentially the same functionality to the ORB. They are used internally by the ORB to make calls up to the object implementation to perform a request and return the results to the ORB.

The static IDL skeleton is generated by an IDL compiler and provides an interface for the ORB to make up-calls to a specific object type. Like the client stubs, they provide functions that allow the ORB to call specific methods of the target object.

The dynamic skeleton interface provides more flexibility and allows the ORB to make up-calls to any object regardless of type. The ORB can construct a request from information received across the network and pass it up to an object using the dynamic skeleton interface.

The object receives the up-call from the ORB and processes it according to its implementation. Any return values are passed back through the ORB to the client, using the same mechanism described previously.

2.2 Component-Based Software Development with Java Beans

Components are self-contained elements of software that can be controlled dynamically and assembled to form applications. Components must interoperate according to a set of rules and guidelines. A container provides a context in which components can interact with each others. The model allows users to construct application by piecing together components.

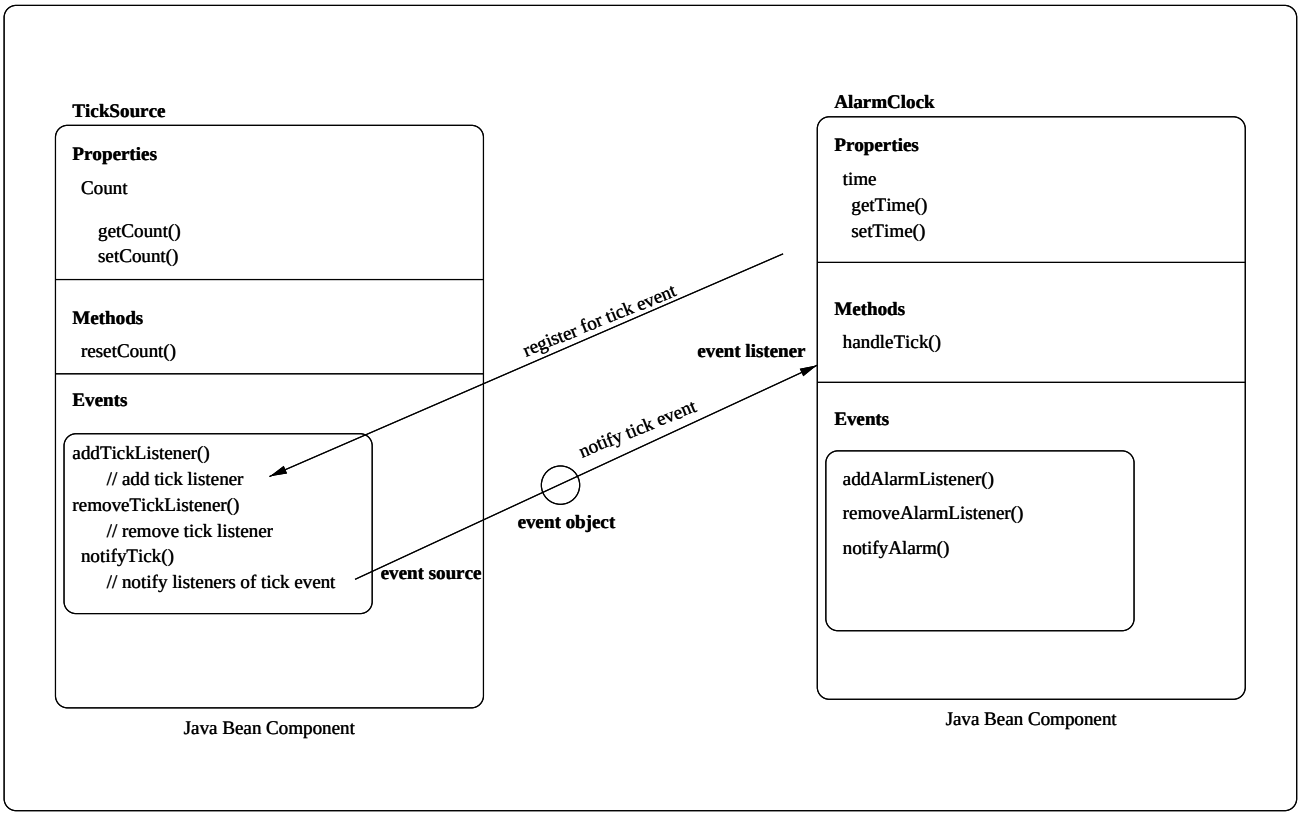
JavaBeans is an architecture for both using and building components in Java. This architecture supports the features of software reuse, components models and object orientation. The following are some of the important features of Java Beans

Properties, Methods, and Events

In order for a component to be reusable, it must allow customization and modification to at least some of its properties. A component must have a behavior and must be able to communicate with other components.

As a software component, each Bean has its own properties. Properties are named attributes that can be read and written by calling methods on the Bean which are specifically created for that purpose. The methods of a Bean correspond to its behavior. The exposed methods represent the interface used to access and manipulate the component. Events are what connects components together. They are the mechanisms used by one component to send notifications to another. One component can register its interest in the events generated by another. Whenever the event occurs, the interested component will be notified by having one of its methods invoked.

Java Beans Component Model



In the above example, 'count' is a property which can be read and written using the 'getCount' and 'setCount' methods. 'resetCount' is an exposed method which can act as a target method for events. 'tick' is an event. The AlarmClock component registers with the TickSource component as a listener for tick event. Whenever the 'tick' event occurs, the TickSource notify the AlarmClock listener by invoking its 'handleTick' method.

Introspection

For components to be reusable in development environments, there needs to be a way to query what a component can do in terms of the methods it supports and the types of event it raises and/or listens for.

Introspection is the process of exposing the properties, methods, and events that a JavaBean component supports. This process is used at run-time, as well as by a visual development tool at design-time.

Customization

The degree to which you can reuse a component is often based on how easily you can customize that component to fit different application requirements.

The properties of a bean can be customized through the use of a property editor generated dynamically by an application builder tool by determining the properties that a Bean supports. Each Bean can provide its own customizer class that can help the user to customize an instance of that Bean.

Persistence

A persistence scheme permits a component to save its state and, subsequently, for the component to be recreated. Components must be able to participate in their container's persistence mechanism so that all components in the application can provide application-wide persistence in a uniform way. Currently, the best and simplest way to support persistence of Java Beans is through the use of Java Object Serialization.

2.3 Frameworks and Design Patterns

A framework captures the programming expertise necessary to solve a particular class of problems. Developing a framework differs from developing a stand-alone application. A successful framework solves problems that appear different from the problem that justified its creation. Adapting the framework to new problems requires an understanding of the solution the framework provides and how it can be used to solve new problems. So it is critical to follow good software design practices while developing a framework. A framework that addresses its design issues using design patterns is far more likely to achieve high levels of design and code reuse. Mature frameworks usually incorporate several design patterns. A design pattern names, abstracts, and identifies the key aspects of a common design structure that makes it useful for creating a reusable object-oriented design.

Chapter 3
System Analysis and Design

3.1 Domain Analysis

This work is being done as part of the Earth Observing System (EOS) project at the College of Oceanic and Atmospheric Sciences, Oregon State University. EOS is part of NASA's effort to study global climate change. The project represents a continuing effort by scientists to carry out studies on a variety of Earth and Space Science data sets collected over periods of time, analyze these data sets and thereby infer changes in global climate. The data sets represent different kinds of data. Some example data include numerical data from the ocean recorded by sensor instruments called drifters that float in the ocean and record various parameters like sea surface temperature, chlorophyll content etc., satellite imagery taken by polar-orbiting satellites that cover the world ocean. Three dimensional data sets include parameters of the ocean like salinity, pressure and temperature at various depths in the ocean recorded by the National Oceanic and Atmospheric Administration (NOAA).

With the expanding availability of a wide variety of data sets in the scientific domain, it is becoming increasingly difficult to manage these data sets and to provide meaningful access to facilitate an analysis and understanding of these data sets. There is basically a need for a framework for building an interactive data exploration system. Doing so requires understanding the interactive and iterative process of data exploration. Essentially, there are three kinds of exploration subtasks.

1. Data manipulation operations: include selecting data for display, focussing on particular attributes of the data and grouping or reorganizing data.
2. Data visualization operations: include presenting or displaying the data in a suitable manner such as effective graphics.
3. Data analysis operations: include statistical analysis, summarization and transformation for understanding properties of the data.

There are several approaches to providing data manipulation operations. One of the popular approaches is to provide a text-based query system wherein the user directly queries the database for retrieving the data he/she is interested in. They provide various levels of support for accessing, modifying and reorganizing data. The main drawback of such approaches are that they are not well integrated with effective graphics techniques. They expose the organization of the data in the database and require the user to be aware of it and make queries based on the organization. Also, in case of geographically distributed databases, it is up to the user to access different databases to retrieve the data he/she is interested in.

When dealing with large data sets, it is not sufficient to be able to retrieve the data and present it textually or create a simple display. Effective visualization and analysis of large data sets requires a

system where in the user has at his disposal a set of coherent, interoperable, interactive and customizable tools. Ideally, the tools should allow the user to be able to customize the analysis to his specific needs. For example, in case of temporal data sets, the user may wish to combine or group different data sets to get a consistent and comprehensive view. Also, the three kinds of exploration subtasks are interdependent and overlapping. For example, a user may wish to select data directly from a visualization (a data manipulation operation performed on a visualization). So, the tool interfaces should be able to support interdependent subtasks.

3.2 Requirements Analysis

The next step involved in developing the system is to formally specify the requirements of the system. In this section, we describe the important high-level system requirements which form the basis for defining the architectural design of the system.

1. Uniform Data Access

The system should provide a well-defined and open interface for uniform access to the data sets. The organization of the data sets in the data stores should be hidden from the client applications. The interfaces should only deal with high-level queries which are independent of the type and the organization of the data stores. The client applications should use these interfaces to retrieve data sets and provide data exploration functionality. CORBA provides an excellent mechanism for specifying and distributing such an interface. An IDL interface to our data stores allows anyone to create a CORBA-compliant application that can use our data stores. By providing a standards-based interface to the data sets, we make the development of client application software easier and cheaper.

2. Data Processing

The data obtained from the data stores may need to be processed or combined with other data sets in order to render it useful for meaningful analysis. The processing may be intensive depending on the amount of the data to be processed and the computational complexity of the algorithm to be used for processing. The system should provide software and hardware resources for efficient execution of such processing tasks.

3. Data Visualization Environment

The purpose of a data exploration system is to enable users to uncover and extract relationships hidden in large data sets. This usually requires flexible data manipulation mechanisms which provide a complete coverage of the operations needed by the users. The system should provide a data visualization environment for the users. Such an environment should essentially provide a set of tools which provide visualization and analysis of the data sets. The environment should enable the users to form adhoc groupings of data so as to facilitate understanding of the relationships latent within the data.

4. Scalable and Extensible

The system should be extensible so that addition of new data sets to the data stores requires only addition of new components to the existing system; it should not require any changes to the existing components. The system should be scalable with respect to the number of clients using the system and the amount of data transfer taking place over the network.

5. Access Control

Controlling access to the data is another requirement. We may want only authorized users to be able to access the data sets. We therefore need to verify that users are legitimate clients before granting them access to the data. We need to implement some kind of security that will ensure that access to the data can be strictly controlled.

6. Performance

Scientific data visualization systems usually require bulk data transfer and involve intensive data processing. The system should have a reasonable response time. The performance of the system should not degrade as the number of clients and the amount of data transfer increases.

3.3 System Design

The system is designed as a three-tier application with a presentation tier, an application server tier and a database tier. All the data resides in the database tier. The database tier may consist of a single database or multiple, geographically distributed databases. The application server tier would provide a set of interfaces for accessing these data sets and also provide any data processing functionality that may be needed. The presentation tier would use the application server interfaces to retrieve the data sets and provide data exploration functionality to the users. We need to define an architecture for retrieving the data sets from the database, processing the data, passing it on the clients and visualizing it in a way that satisfies our requirements. We'll explain some of the design issues and solutions to them which would help us define the architecture of the system.

3.3.1 Design Issues

1. Distributed Data Processing

One of the main tasks of the system is to retrieve data from the data stores based on client queries, process it and hand it over to clients.

The performance of the system depends on the following factors:

1) the amount of the data to be retrieved from the data stores

1) the amount of data to be processed

2) the computational complexity of the data processing algorithm

3) the number of clients using the system simultaneously

The performance should be enhanced. CORBA provides a good distributed object infrastructure which can be used for distributed computing to boost the performance of the system. The data retrieval and processing functionality could be distributed among multiple distributed CORBA objects running on different machines to achieve a performance gain.

The processing task consists of: 1) an algorithm, and 2) the data which has to be processed in accordance with the algorithm. The algorithm may be a static server-side algorithm or a dynamic client-specific algorithm which is known only at run time. In order to support both types of algorithms, the data processing CORBA server should be able to accept and execute different tasks. A task-divider

process will divide the task into sub-tasks, each of which consists of a portion of the data and the algorithm to be applied to the data. Each of these sub-tasks is migrated to a CORBA object which will execute the task and return the results back to the task-divider process. The sub-results will be combined into the final result.

A general example would be an array sorting algorithm such as quicksort. In this case the algorithm is the quicksort algorithm itself and the data is the array of integers to be sorted. The task-divider process would partition the original array into two in accordance with the quicksort partitioning algorithm. Each of these partitions form the data for the sub-tasks. The sub-tasks are executed concurrently on distributed machines and the result of each subtask which is a sorted array is sent back to the task-divider process. The task-divider process combines the sub-results to form the final result.

2. Visualization Environment

An essential requirement is that the visualization environment should provide flexible data manipulation mechanisms. A client should have at his disposal a set of collaborative tools. Each tool should provide a complete set of operations needed by clients for visualization and analysis of a particular type of data set. These tools should be able to flexibly interact with each other to support groupings of data. A client should be able to customize the environment according to his visualization needs and should also be able to save the state of his environment for later use.

A very good way to provide such an environment is through component-based software development. Components are self-contained reusable elements of software that can be controlled and assembled dynamically to form applications. We can develop components for visualization of each type of data set. These components are highly customizable. They could be assembled in a builder-tool and could also interact with each other at runtime. The persistence of these components would be used to save the state of the visualization environment.

3. Persistence

The issue of persistence is: How are we going to store the data that we need to make the system work? There are two types of data to be stored:

1) The database of user IDs and password. We will use Java serialization to persist the user information into files, while making sure to use a generic interface internally to allow us the flexibility to upgrade to another scalable implementation in the future such as storing the user ID database in an SQL table.

2) The user's workspace which will consist of a set of components. The components would be persisted into streams using Java serialization and stored in files on the servers or in the database.

4. Security

Security is another issue that we need to address. Security is a broad term that covers a wide range of issues. Usually it refers to mechanisms for authentication, authorization, and encryption of data. For our purposes, we are concerned only about authentication because we want to block convenient access to the data to unauthorized users. We use a simple mechanism: Once a user is authenticated as a legitimate client, we will give him or her full access to the data. Our implementation will use a simple password scheme to authenticate the user and rely on the passing of an authentication token in subse-

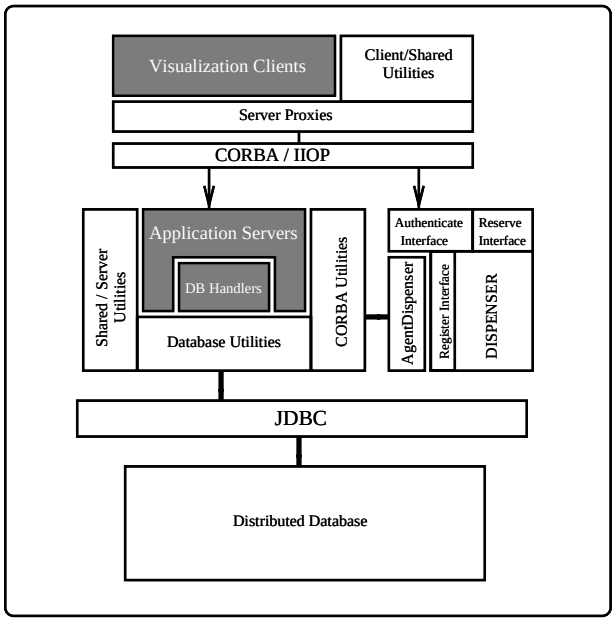
quent operations to verify that user’s identity. Using an encryption mechanism in the authentication process would be advisable.

Chapter 4
A Distributed, Three-tier Framework

4.1 Three-tier Framework Description

We have a three-tier framework with the client visualization components in the presentation tier, the application servers in the middle tier and the data stores in the third tier. The application servers run as distributed objects on multiple servers in a heterogeneous environment and embody most of the logic related to data retrieval and computation. Thus, the client or the front-end becomes very thin, making it suitable to run on low-end machines such as lightweight PCs and the JavaStation. The client can run either as an application or as an applet.

The innovative part of our framework is the way in which the client components interact with each other and with the server objects and how the server objects distribute the process of data retrieval and computation among themselves to provide a flexible, high-performance environment. The application servers run as distributed CORBA objects. Each type of server provides uniform access to a particular type of data through the use of a database handler which encapsulates database specific querying logic. In addition, it can also perform computation intensive tasks efficiently by distributing the task among a set of available computation servers running on a High Performance Cluster (HPC). Several instances of each type of server may be running on multiple heterogeneous machines.



System Framework

The front-end consists of a set of client components, each of which acts as a visualization tool for a particular type of data, and a set of filter components which are used by client components for various types of data manipulation. The client components communicate with the server objects for data retrieval through the use of server proxy objects which encapsulate the calls needed to contact the server objects and invoke methods on them making the client components independent of the implementation of server objects. These client components implement interfaces allowing them to interact with each other to provide a basis for data exploration. Such interactions may be direct communication between client components involving transfer of data or may be indirect through a filter component wherein data is 'filtered' before being passed on to the other component.

4.2 System Components

4.2.1 The *Dispenser* :

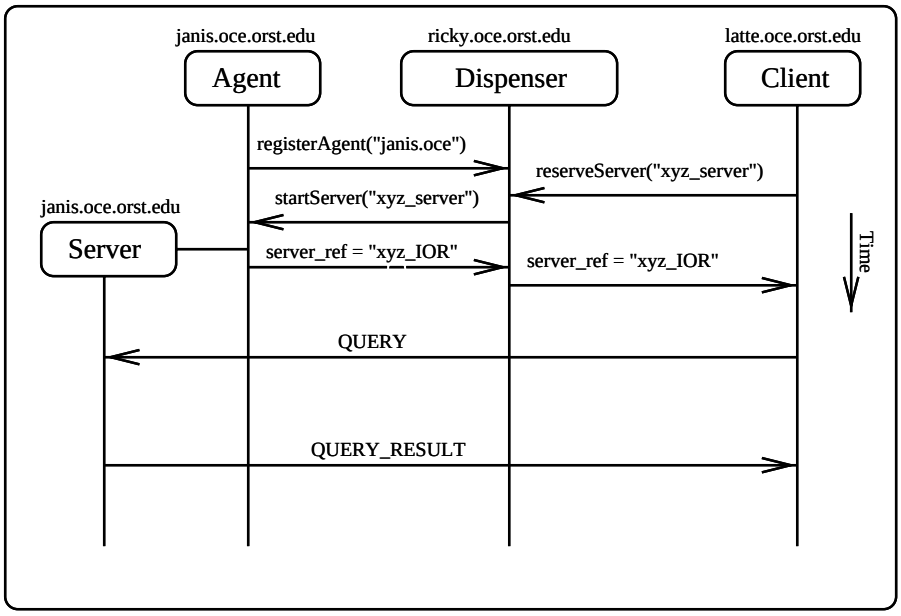
The Application Server tier (middle tier) consists of a set of CORBA objects running in a distributed heterogeneous environment. These server objects being CORBA objects can be written in any of several languages having CORBA support and may use any of the available ORBs. The client components in the front-end communicate with these server objects for data retrieval. The *Dispenser* is a CORBA object which creates and manages these server objects and acts as a registry which the clients can use for accessing a reference to the server objects.

However, the Dispenser by itself would only be able to create servers which run on the same machine as the Dispenser. This is because the current implementation of CORBA does not enable a CORBA object to start another CORBA object on a different machine. But in a distributed environment we would like different servers to run on different machines and make utmost use of the resources. We could manually start a CORBA server on any machine and it could go and register with the Dispenser making itself available to the system. However, from the administration point of view this is not a good idea because as the number of distributed servers increase, managing these would be a problem. We would like the system to be able to dynamically start servers on any of the machines available for computing and also do some load balancing.

We use the concept of an 'Agent Dispenser' to achieve this. An instance of an 'Agent Dispenser' (hereafter referred to as Agent) runs on every machine which is available to the system. This Agent registers with the Dispenser when it is started. The Agent itself acts as a factory for the creation of servers. Each Agent maintains a registry of all the servers running on its machine. The Dispenser maintains a registry of all the agents and all the servers running under each agent. Whenever a Dispenser wants to start a server, it can dynamically send a request for starting the server to the Agent running on that machine. The Agent starts the server and makes the server's identity available to the Dispenser. The Dispenser can do some load balancing by deciding which machine to start the server on by using the number of servers running on each machine as a load factor.

The following figure shows the the interactions between the Dispenser and an Agent when a client needs to reserve and communicate with a server. When the agent is started on a machine, it

accesses the Dispenser and uses its AgentDispenser interface to register itself with the Dispenser. Now, when a client sends a request to the Dispenser for reserving a server "xyz_server", the Dispenser contacts the agent running on the machine on which it wants to start the server and requests the agent to start the server "xyz_server". The agent start the server locally and returns the server's reference to the Dispenser. The Dispenser returns the server's reference to the client to complete the reserveServer operation initiated by the client.



Agent - Dispenser Interaction Diagram

The Dispenser implements the following interfaces:

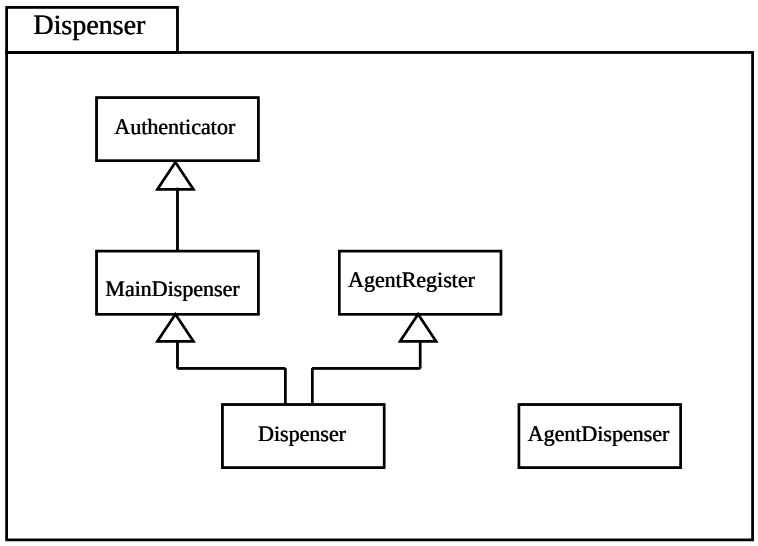
Authenticator Interface: Clients need to be authenticated before being allowed access to server objects because the server objects provide direct access to the database.This interface lets a client authenticate itself with the Dispenser allowing it to request a reference to a server object. The Dispenser maintains a registry of authorized users.

MainDispenser Interface: The MainDispenser interface is used by clients to get CORBA references to server objects. The scheduler incorporates load balancing logic to select a server object with minimum load in case multiple instances of that server object are registered. The MainDispenser interface extends the Authenticator interface. This is the interface which is used by the clients to authenticate themselves into the system and use the Application Servers.

AgentRegister Interface: This interface is used by the Agents to register themselves with the Dispenser. An Agent registers its name (its unique IP address) and its reference (Interoperable Object Reference (IOR)) with the Dispenser. This interface is not visible to the clients. This interface can also be used by server objects to register with the Dispenser. This may be useful to be able to start server objects on a machine without starting any Agent Dispenser on it, or to start server objects written in different languages.

Dispenser Interface: This is the interface which is actually implemented. It extends the MainDispenser and the AgentRegister interfaces. The clients get a reference only to the MainDispenser interface, while the Agents get access only to the AgentRegister interface. This makes the client transparent to the internal dispenser implementations such as the Agents, etc.

AgentDispenser: This interface represents an Agent. It provides operations for starting a server on the machine on which the Agent is running.



Dispenser Class Diagram (UML)

The IDL definition of the Dispenser:

```
module DispenserModule
        {
        typedef sequence<octet> AuthToken;
        struct AuthPacket
                        {
                        string userID;
                        AuthToken token;
                        };
        exception AuthFailure
                        {
                        string reason;
                        }

        interface Authenticator
```

```
                {
                AuthPacket login(in string user, in string passwd)
                        raises (AuthFailure);
                void logout(in AuthPacket packet)
                        raises (AuthFailure);
                };

        exception DispenserException
                {
                string reason;
                };

        typedef sequence<string> Servers;

        interface MainDispenser  : Authenticator
                {
                Servers getListOfServers(in AuthPacket token);
                boolean isServerAvailable(in AuthPacket token, in string server);
                any reserveServer(in AuthPacket token, in string server)
                        raises (DispenserException);
                void releaseServer(in AuthPacket token, in any server)
                        raises (DispenserException);
                };

        interface AgentRegister
                {
                void registerAgent(in string agent, in string ior);
                void registerServer(in string server, in string ior);
                };

        interface Dispenser : MainDispenser, AgentRegister
        {
        };

        interface AgentDispenser
                {
                string startServer(in string server);
                boolean stopServer(in string server, in string ior);
                };
}
```

4.2.2 Application Server Objects:

A set of Application Servers running in the middle tier provide uniform access to all the data sets. A large system will have several application servers. A critical design issue is: How do we decide what goes into each Application Server? The partition should be such that it allows a part of the application to be upgraded without affecting the other parts. However, a very fine-grained partition may result in increased communication between servers and would reduce performance if remote method invocations are involved. Therefore, we must take care to partition the application so that as few remote method invocations as possible are necessary.

The system partitions the application layer based on the data sets.

1. Each Application Server should provide a complete set of operations for access to a particular type data set.

2. Application Servers may collaborate with each other; however such collaborations should not stem from the requirements of a client's data manipulation mechanisms. This enables Application Servers to be fairly independent of Client Components and at the same time allows new Client Components to be developed without any need for extra support from the Application Servers which would otherwise require changes to Application Servers.

An example application server:

```
module EOSModule
        {
        interface EOSServer
                        {
                        boolean isAvailable(string server);
                        };
        }

module DrifterModule
        {
        exception DrifterException
                        {
                        string reason;
                        };
        struct Region
                        {
                        double northLatitude;
                        double southLatitude;
                        double eastLongitude;
                        double westLongitude;
                        };
        struct TimeFrame
                        {
```

```
                string fromDate;
                string toDate;
                };
struct Drifter
                {
                long id;
                };
typedef sequence<Drifter> DrifterSeq;
typedef string Value;
struct DrifterQuery
                {
                Region region;
                TimeFrame timeFrame;
                DrifterSeq drifter;
                Value value;
                };
struct DrifterData
                {
                Drifter drifter;
                double latitude;
                double longitude;
                double value;
                string date;
                };
typedef sequence<DrifterData> DrifterDataSeq;

interface DrifterServer : ::EOSModule::EOSServer
                {
                DrifterSeq getDrifterIDs(in AuthPacket token, in Region region)
                                raises (DrifterException);
                DrifterDataSeq getDrifterData(in AuthPacket token,
                                                in DrifterQuery query)
                                raises (DrifterException);
                };
};
```

4.2.3 Server Proxy Objects:

A *Server Proxy*, associated with every *Server Object*, encapsulates CORBA-specific code required for communicating with the *Server Objects*. Upon creation, a Server Proxy gets a reference to its Server Object which it will use for further communication with the Server Object.

Server Proxy Objects are used by client components to communicate with *Server Objects* .

Clients are programmed to use Server Proxy Interfaces which specify operations for communication with the middle tier. These operations correspond to the operations specified in the interface of Server Objects. Server Proxy Objects represent implementations of these interfaces and these implementations depend on the architecture used for the middle tier. For example, for a RMI-based middle tier, the Server Proxy Objects implement the Server Proxy Interfaces using the RMI mechanisms. The goal is to make the client component transparent to the network communication mechanism used for communication with the middle tier. This essentially insulates the presentation tier from changes to the implementation of the application tier.

4.2.4 Database Handlers :

Every *Server Object* uses a *Database Handler* for communicating with the database. Different data sets may be residing in different types of databases. Also, the type of database may change over time. However, we would like our Application Server Objects to be independent of the type of database used to store the data. Hence, the design issue is: How can the application servers interact with the database without depending on the type and location of the database? The solution is to define standard interfaces against the database, and implement these interfaces using database specific logic. This essentially decouples an abstraction from its implementation, allowing the implementation to vary independently. (BRIDGE design pattern)

We use the Java Database Connectivity API for connecting to the database and querying it. A *Database Handler* converts Client queries into database specific queries and returns the results back to the *Server Object*.

Database Connection Management:

The number of simultaneous connections to the database is usually limited. If every Database Handler creates its own set of connections to the databases then the total number of simultaneous connections would depend on the number of Database Handlers and may exceed the limit. Hence the database connections should be managed separately and should be shared by the Database Handlers. A system-wide DB Connection Pool is maintained which provides a set of connections and a set of operations for reserving and releasing a connection.

Sample Code:

```
public class ConnectionPool
{
    protected static int numDB = 2;
    protected static int numCons = 2;
    protected Hashtable conPool;

 static
  {
   String url[] = new String[numDB];
   String user[] = new String[numDB];
   String password[] = new String[numDB];
   url[0] = "jdbc:weblogic:sybase:cupcake";
```

```
  user[0] = "guest";
  password[0] = ""
  url[1] = "jdbc:weblogic:sybase:suzyq";
  user[1] = "guest";
  password[1] = ""

  conPool = new Hashtable();
  for (int i=0; i<numDB; i++)
    {
       DBConnection[] cons = new DBConnection[numCons];
       for (int j=0; j<numCons; j++)
         {
           cons[j] = new DBConnection();
           cons[j].con = DataBaseUtils.connect(url[i],user[i],password[i]);
           cons[j].url = url[i];
           cons[j].user = user[i];
           cons[j].password = password[i];
           cons[j].inUse = false;
           cons[j].number = j;
           }
         conPool.put(url[i],cons);
     }
}

public static synchronized DBConnection reserveConnection(String url)
{
  DBConnection[] cons = (DBConnection[]) conPool.get(url);
  for (int i=0; i<numCons; i++)
    {
         if ( !(cons[i].inUse))
           {
             cons[i].inUse = true;
             return cons[i];
           }
    }
  return null;
}

public static synchronized void releaseConnection(DBConnection con)
{
  DBConnection[] cons = (DBConnection[]) conPool.get(con.url);
```

```
    cons[con.number].inUse = false;
  }
}
```

4.2.5 Computation Server Objects :

A *Computation Server Object* assists an *Application Server Object* in performing computation-intensive tasks. A *Server Object* can distribute its task among multiple *Computation Server Objects* depending upon the computational complexity of the task and the number of *Computation Server Objects* available at runtime.

Design Issue: The computation task essentially comprises of an algorithm and the data to be processed. The algorithm has to be made available to the computation server dynamically at runtime along with the data retrieved from the database. Since the computation servers are CORBA objects running in a distributed environment, the task has to be shipped across the network in order to execute it on the processor on which the computation server is running. However, CORBA does not support object by value parameter passing.

Solution: We use Java's reflection and serialization mechanism to pass objects by value. This solution only works for Java-based CORBA objects. Every task that is to be executed remotely implements an interface called 'RemotelyExecutableTask' which essentially has two methods: executeTask and getResult. The task object is serialized into a byte stream and sent to the remote Computation Server. The Server extracts the task from the byte stream and casts it into a RemotelyExecutableTask. It invokes the executeTask method which causes the algorithm to be executed. It then invokes the getResult method which returns the results in the form of an object. The Server serializes this result object and returns in the form of a byte stream.

The RemotelyExecutableTask interface:

```
public interface RemotelyExecutableTask
{
      public void executeTask();
      public Object getResult();
}
```

An example task looks like this:

```
public class QuickSort implements RemotelyExecutableTask, Serializable
{
private long[] a;
public QuickSort(long a[])
{
        this.a = a;
}
public void executeTask()
{
        quicksort(a,0,a.length-1)
```

```
}
quicksort(long a[], long p, long q)
{
        long j;
        if (p<q) {
          j=q+1;
            j=partition(a,p,j);
          quicksort(a,p,j-1);
          quicksort(a,j+1,q);
          }
}
public Object getResult()
{
        return a;
}
}
```

The ComputationServer interface is as follows:

```
interface ComputationServer
        {
        octet[] compute(in octet[] task);
        }
```

The "compute" method essentially takes a computation task an an byte stream, executes the task, and sends back the result in the form of a serialized byte stream.
The implementation of the method is as follows:

```
 public byte[] compute(byte[] arr)
{
     try {
     ByteArrayInputStream is = new ByteArrayInputStream(arr);
     ObjectInputStream ois = new ObjectInputStream(is);
     RemoteTask task = (RemoteTask)ois.readObject();
     task.executeTask();
     java.lang.Object result = task.getResult();
     byte[] res = Serializer.serialize(result);
     return res;
    }
    catch (Exception e) {
     System.out.println(e);
    }
```

}

Design Issue: The main Server Object has make simultaneous method invocations on multiple Computation Servers so that Computation Server can be run concurrently.

Solution 1: Using Java's multithreading

Java's multithreading capability help a Server Objects to invoke multiple Computation Server Objects simultaneously by invoking each of them in a separate thread.

Solution 2: Using Distributed Callback Interface

As an alternative to using threads for making simultaneous invocations, distributed callback interfaces can be used. This solution uses CORBA's "oneway" methods for making asynchronous invocations and requires the main Server Object to define a callback interface which will be used by the Computation Server to send back the results when they are available.

The Computation Server's 'compute' method can be converted into a asynchronous call by using the 'oneway' method feature.

```
interface ComputationServer
        {
        oneway void compute(in octet[] task, in CS_callback callback_reference);
        }
```

The main Server Object has to implement a callback interface and send its reference to the Computation Server along with the task to be executed. The Computation Server will use the Server Object's callback interface to send back the results when they are available.

The callback interface looks like this:

```
interface CS_callback
        {
        oneway callback(in octet[] result, in boolean successful);
        }
```

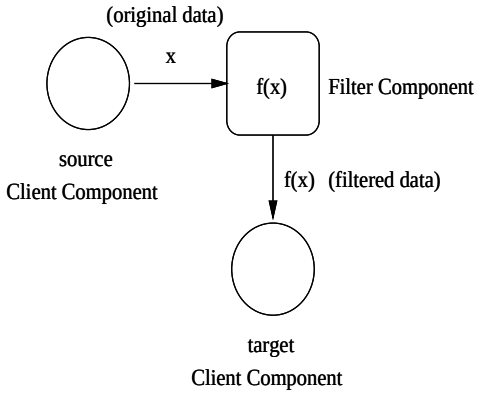
4.2.6 Client Workspace :

The *Client Workspace* is basically a Java Beans container providing a context in which the *Client Components* can interact with each other. In our implementation, the *Client Workspace* is a customized version of the the Java Bean Box container that is part of the Beans Development Kit (BDK). We have customized the BeanBox so that it runs as an applet in the browser. The *Client Workspace* supports remote serialization allowing a client to save his workspace into the remote database for later use.

4.2.7 Client Components **and** *Filter Components* :

A *Client Component* is a Java Bean Component which runs in the *Client Workspace*. It interacts with the user and can be linked with other components in the *Client Workspace*. It communicates with one or more *Application Server Objects* to retrieve its data. A *Filter Component* is also a Java Bean Com-

ponent which runs in the *Client Workspace*. It is meant to be used in conjunction with *Client Components* to produce filtered versions of datasets. It is associated with a source *Client Component* and a target *Client Component*. It takes data from the source, filters/manipulates the data according to some algorithm implemented by it and returns the data to the target. A source component can itself be the target of a *Filter Component*. The framework allows a user to define his own filter and use it in the workspace.



Client and Filter Component Interactions

4.3 Technical Issues

Designing and developing a distributed framework is a difficult task. There are many technical challenges related to functionality, availability, reliability, security and performance which need to be addressed. In this section, we briefly describe some of the design issues related to the proposed framework and show how they are addressed.

1. High Availability

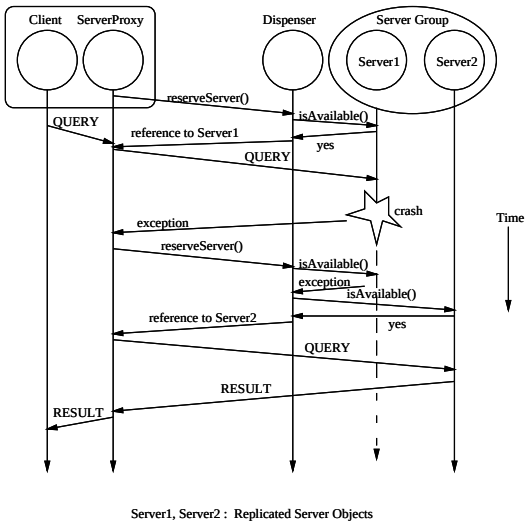
The front-end Client Components rely on the Application Server Objects for data retrieval. Non-availability of Server Objects would result in inaccessibility of the database. One technique for improving Server availability is replication of Server Objects implementing the same interface to form an object group. In our system, multiple Server Objects of the same type may be running in a distributed environment.

2. Reliability

Distributed applications often require substantial effort to achieve levels of reliability equivalent to those expected from stand-alone applications. In our system, an Application Server Object might become unreachable after it is created by or registered with the Dispenser. However, the Dispenser continues to retain the Server's reference assuming its availability. So, detecting failures of Server Objects is needed. In our system, every Application Server Object implements a method called isAvailable() which returns a boolean true. When a client request for a Server reference comes in, the Dispenser invokes the Application Server Object's isAvailable() method to determine its availability.

The Server Object would return true if available, otherwise an exception would be thrown by the ORB, stating the non-availability of the implemented object, which would be caught by the Dispenser. If the Server Object is not available, the Dispenser removes the Server from its registry and tries for other Server Objects of the same type.

The second problem is that an Application Server might crash while it is processing a client's request (fig 2). This would result in an ORB throwing a communication failure exception. In our framework, the exception is caught by the Server Proxy which immediately accesses the Dispenser for a new reference to the Server Object. The Dispenser realizes the failure of the first Server Object when it invokes its isAvailable() method and so returns a reference to a replicated Server Object. The Server Proxy reinvokes its request on the new reference to the Server Object and proceeds to hand over the results to the Client. Thus, the Client is totally transparent to any crash of an Application Server Object.



Reliability of the Middle Tier

The other problem is that the Dispenser itself may crash. In this case, the problem is that the Application Server Objects which have been created by or registered with the Dispenser have no idea about it. So, they would not try to register again with the Dispenser when it becomes functional. To overcome this problem, the Dispenser serializes its registry (using Java's object serialization) into a file every time an Application Server Object registers with it. Whenever the Dispenser starts, it checks for the registry file and reloads any serialized data available in it.

3. Interoperability

Every ORB provides its own implementation of the CORBA COS (Common Object Services) naming service which can be used to store and retrieve object references. However, these nameservice implementations are not interoperable; that is, objects written using one ORB cannot register with or look into the nameservice of another ORB implementation. For example, if an object A which uses the Java IDL ORB needs to communicate with an object B which uses the VisiBroker ORB and B is

registered with the VisiBroker's directory service *osagent*, the Java IDL object A cannot access the *osagent* to get a reference to B. Inclusion of the Dispenser object in our design is mainly to resolve this issue.The Dispenser object implements a register interface which acts as a common nameservice allowing objects written using different ORBs to register with it so that they are accessible by any client in the framework.

Chapter 5

Example Application and Performance Measurements

In this section, we discuss the working of a particular application we have developed using the proposed framework and briefly look at some performance measurements. The application consists of the following Application Server Objects and Client Components:

5.1 Application Server Objects:

1. *DrifterServer* :

A drifter is a physical sensor instrument that is dropped in the ocean so that it moves along with the currents, recording its coordinates and ocean parameters like sea surface temperature, sun angle, etc. The DrifterServer provides access to the drifter data such as the position of a drifter and values measured by it. The data set is visualized as a two-dimensional data with the latitude and longitude coordinates forming the two dimensions. The 2-D plot is color-coded based on the parameter value that is being displayed.

Query and QueryResults:

1. Instrument Ids based on region:

This simple query is used to retrieve all the instrument Ids that have data in the specified region. The query consist of a region frame in terms of latitude and longitude. The result consists of an array of instrument Ids.

2. Drifter Tracks:

This is the query which actually retrieves the drifter parameter values which are used to generate a track. The query consists of the following attributes:

a) Region in terms of latitude and longitude

b) Time Frame - start data and end date

c) Drifter Sequence - the set of drifters (instrument Ids) whose data is to be retrieved.

d) ValueType - the parameter whose values are to be retrieved (such as sea surface temperature, sun angle, etc.)

The query result consist of the following columns

a) Drifter - The instrument ID

b) Region (latitude and longitude) - the position of the drifter

c) Value - the recorded value

d) Date - The date and time at which the value was recorded

2. *CoastlineServer*:

This data set represents the coastline on the earth. The CoastlineServer provides access to the coastline data corresponding to a given region.

Query and QueryResults:

1. Coastline based on Region:

The query consists of a region for which the coastline data is to be retrieved. The result consists of the coastline coordinates for the queried region.

3. *ImageServer*:

This data set consist of the satellite imagery taken by polar-orbitting satellites that cover the world ocean. These images are 8 bits/pixel images and the pixels encode the temperature of the ocean at that coordinate. The ImageServer provides access to the satellite imagery.

4. *ImageProcessorServers*:

These are the computation servers used by the ImageServers for processing images retrieved from the database. The main image processing that is done is to overlay the drifter data on the images in order to generate a time based animation. The processing involves data intensive computations which include decompressing a GIF image, manipulating its pixel values and/or color map and compressing it back into the GIF format.

5. *MatlabServer:*

The MatlabServer is used to provide Matlab plots of the data retrieved from the data stores. The MatlabServer is written to use the native Matlab libraries and the C API provided by Matlab. When the server receives a request for Matlab, it start the Matlab engine and invokes operations on it to do computations and generates the results in the form of plots which are returned to the client.

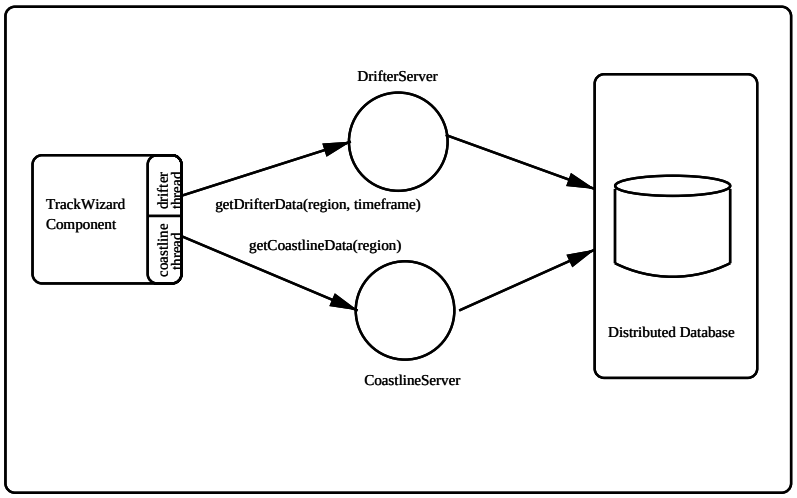
5.2 Client Components

1. *TrackWizard*:

This component communicates with the DrifterServer and the CoastlineServer, and provides a two dimensional plot of the track taken by a drifter (or a set of drifters) in the ocean along with the coastline corresponding to that region. Java's built-in multithreading helps in retrieving both the drifter data and the coastline data in parallel by communicating with the DrifterServer and CoastlineServer simultaneously in different threads.

The DrifterServer and CoastlineServer running as separate processes in a distributed environment

execute the request concurrently resulting in the drifter data and the coastline data being retrieved in parallel. The Servers return the data to the TrackWizard component which is responsible for visualizing it appropriately.



Concurrent Processing of Requests

2. *ImageWizard*:
This component communicates with the ImageServer and provides display and animation of satellite imagery.

3. *AlgoBean*:
This is a Filter Component which modifies the drifter data according to an algorithm. The TrackWizard could be its source components. The target components could be the ImageWizard or the TrackWizard itself.

5.3 Distributed Image Retrieval and Image Processing

In this section, we describe an example which clearly demonstrates the use of multiple, distributed computation servers for maximizing parallelism in data parallel computations.

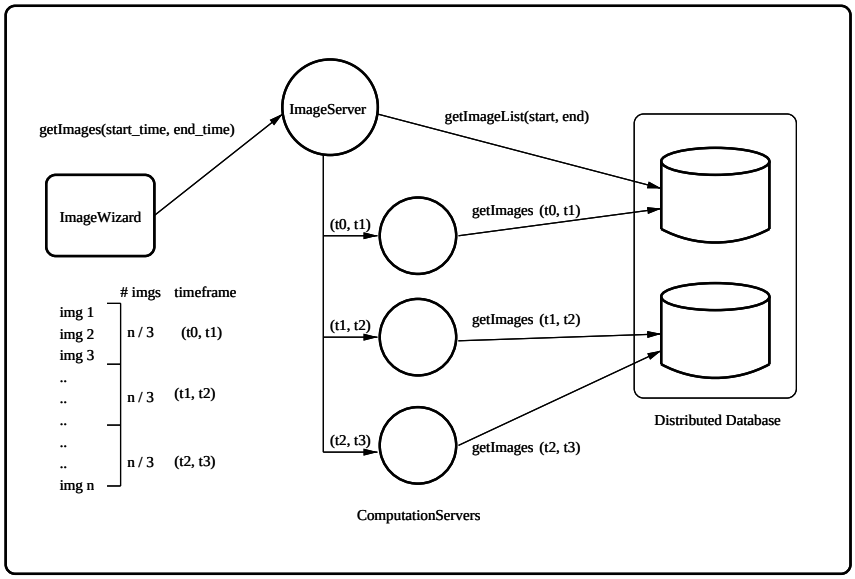
In this example, the main task is to retrieve a set of images from the database based on time frame and process these images. The processing consist of overlaying the drifter tracks on these images so as to be able to produce an animation of the images which shows the motion of the track.

The main task has to be divided into subtasks which can be executed in parallel. The question is how to effectively divide the task into subtasks. The point to note is that the task essentially is to retrieve and process an set of images. Assuming that time to retrieve and process each image is the same for all the images, the total time would be the retrieval and processing time of one image multiplied by the

number of images.

total time = number of images *
(time to retrieve one image + time to process an image)

So, the task division could be based on the number of images to be retrieved and processed. If there are "n" images in the query and there are "k" computation servers being utilized for concurrent processing, each subtask would consist of retrieving and processing "n/k" images.



Distributed Image Retrieval and Processing

The task division is done as follows. The ImageServer upon receiving the query from the ImageWizard, generates a query to retrieve a list of all the image names which have to be retrieved as a result of the query. It then divides this into sublists based on the number of computation servers. For each sublist, it notes the time frame and generates a new query based on that time frame. These new queries form the subtasks for the computation servers.

Another important point to note in this example is that each image can be processed independently of the other images retrieved in the same query. So, the processing of images can be performed in parallel to image retrieval. This can be achieved by running the image retrieval and image processing code in two separate threads. However, there are certain subtle implementation issues which are to be considered. Note that, after the images are retrieved and processed, they are stored at the server and the URL of these images is given to the client. The client is responsible for downloading these images and generating an animation. If the retrieved images are kept in memory until they are processed, then there is the problem of the Java Virtual Machine running out of memory if images are retrieved faster

than they are processed which is normally the case. There are two solutions to this problem. The first one is to retrieve the image and store it on the disk. Later, when the image processing thread is ready to process the image, it loads the image from the disk, processes it, and saves it back to the disk. However, this is a very inefficient solution because disk read and write operations are very slow and should be avoided as far as possible. The other solution is to synchronize the two threads (the retrieval thread and the processing thread) so that at any instant an upper limit on the number of images in memory is maintained. This means that whenever the limit is reached, the retrieval thread will be suspended until some of the images are processed.

The second alternative has been used and has been implemented in a manner similar to the famous producer-consumer solution. The image retrieval thread is the producer of images and the image processing thread is the image consumer. The two threads are synchronized through the use of an image reserve. The ImageReserve class represents an image reservoir where images are stored until they are processed. It has an upper limit on the number of images it can hold.

```
class ImageReserve
{
 private static final int MAX_SIZE = 5;
 private Vector images = new Vector();

 public synchronized InputStream get()
 {
  while (images.size() == 0)
   {
       try
        {
         wait();
        }
       catch (InterruptedException e)
        {}
   }
  InputStream tmp = (InputStream)images.elementAt(0);
  images.removeElementAt(0);

  if (images.size() == MAX_SIZE - 1)
   notifyAll();
  return tmp;
 }

 public synchronized void put(InputStream in)
 {
  while (images.size() > MAX_SIZE)
   {
```

```
        try
          {
            wait();
          }
        catch (InterruptedException e)
          {}
    }
   images.insertElementAt(in,images.size());
   if (images.size() == 1)
     notifyAll();
  }
}
```

The 'put' method is used by the producer to store images in the reserve. When the number of images in the reserve reaches the upper limit, the producer goes to sleep by executing a wait. Similarly, the 'get' method is used by the consumer to get images from the reserve. Whenever it takes an image from the reserve, it checks to see if the producer is sleeping and if so, it wakes it up. The consumer goes to sleep if it tries to 'get' an image from the reserve and realizes that the number of images in the reserve is 0.

5.4 Component Interactions

A very good example which demonstrates the flexibility of Java Beans in this application is addressing a user's need for a mechanism by which he could overlay the drifter track on the images based on region and time and view an animation. In the absence of the component model, a naive solution would be to develop a new component for the specific purpose which would retrieve both the drifter data and image data and perform the computation to display the animation. The component model enables us to design the components so that the user can just drag the drifter track from the first component and drop it on the second component, and perform the necessary computation to provide the animation.

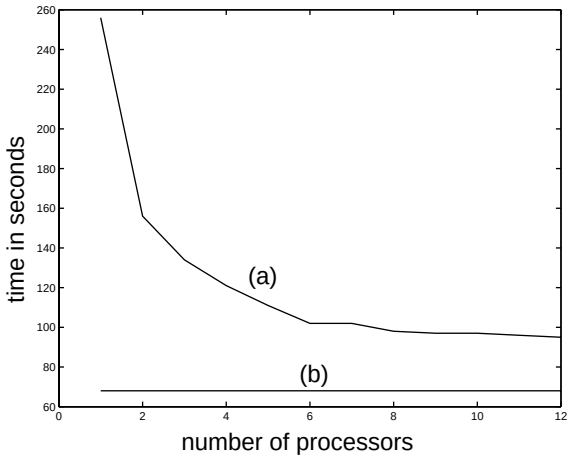
5.5 Performance Measurements

We conducted experiments to calculate the speedup achieved by distributed computing. The experiments we conducted to test the performance of the ImageServer object which basically performs two tasks, viz., image retrieval and image processing. Image processing is a computation-intensive task involving image decompression and compression and hence a suitable candidate for distributed computing. The ImageServer performs the image retrieval on its own, but distributes the image processing task among a set of available ImageProcessorServers. Each image can be processed independently of the other images retrieved in the same query. So, the processing of images can be performed in parallel to image retrieval. The ImageServer divides the total number of images to be processed (say n) among the number of dynamically available ImageProcessorServers (say k). After retrieving every bunch of n/k images, they are handed to an ImageProcessorServer for processing. The

important point to note here is that the image retrieval time remains constant even though the image processing time varies according to the number of available ImageProcessorServers. However, in the experiments conducted we measured the total time which includes image retrieval time and image processing time.

The experiments were conducted on a network of computers consisting of Sun UltraSparcs. The HotJava browser was used to run the clients.

Figure shows the performance curve for the test runs



(a) total time (image retrieval time + image processing time)
(b) image retrieval time

Performance curve: number of processors versus time.

Chapter 6
Summary, Conclusions and Future Work

A Summary of Work

This report describes the design and development of a distributed framework for scientific information management and data exploration. The framework has been designed to address several issues which are typical of such scientific application domains. Prominent among them are:

complexity of managing services and resources in distributed architectures
lack of well-defined, flexible user interface/tools
difficulty in providing distributed, parallel computations
supporting thin-clients for web-based data access;
delegating client-specific computations to back-end services for efficient execution
scalability in terms of number of clients and amount of data access
extensibility to incorporate additional data sets

The design is based on a client/server multi-tier architecture and uses the distributed and component-based models of software development. The application tier provides a set of well-defined services which can be used by clients for accessing distributed data stores and executing scientific computations. The services are implemented as a set of collaborative server objects running in a distributed environment.

The use of CORBA as a framework for distributed object computing is an integral part of the design. CORBA permits the design of open distributed systems in an object-oriented fashion by providing an infrastructure that allows objects to communicate independent of the specific programming language and techniques used to implement the objects. It provides a high-level communication mechanism by use of an interoperable Object Request Broker (ORB) which encapsulates all the low-level details of object communication and object implementations. This allows the application developers to concentrate on solving the business problems at hand.

The CORBA framework has been used to design an architecture which supports distributed, concurrent execution of data-parallel tasks. The design provides a framework for starting services dynamically on remote machines and managing these services and resources in a distributed environment. As the number of clients request increases, more servers can be started dynamically to handle them. New servers can be coded easily to handle new data sets without requiring any changes to the existing servers.

The client visualization tools are composed of a set of interactive Java Bean components which

can run on any platform. These components communicate with the services through the use of server proxies which hide the network communication mechanism and insulate the components from changes to the communication mechanism. These components are customizable at design and run time and can be assembled in container applications to support various types of interactions.

Conclusions

The following is a summary of conclusions:

It is becoming feasible to develop performance-sensitive distributed application in Java:

Java has come a long way since it came into existence. The built-in support for GUI development, multithreading, image processing, automatic memory management and exception handling in Java simplifies the task of developing a distributed systems. The availability of Just-in-Time (JIT) compilers for Java has proved very useful for computation-intensive applications such as ours allowing the system to perform quite well in comparison to C/C++ applications.

CORBA complements Java's capability for developing high-performance, distributed object applications:

Java, being an architecture-neutral, portable and multithreaded programming language, is almost the ideal language for writing portable client/server objects. But it needs an intergalactic distributed object infrastructure to complement its capabilities for high-performance, distributed object computing. CORBA provides a flexible high-level distributed object computing middleware. Pools of server objects can communicate using the CORBA ORB. These objects can run on multiple servers to provide load-balancing for incoming client requests. CORBA's distributed object infrastructure makes it easier to write robust networked objects and coupled with Java's built-in multithreading helps achieve high performance.

Component-based technologies support web-based software development architectures and provide reusability and dynamic configurability:

Ccmponent models such as Java Beans offer the ability to develop software by piecing together reusable and customizable software parts. This ability enables software developers to develop software components on a variety of hardware/software platforms. These components can become a part of a web-based software library which can be used for developing various types of custom applications.

Java has certain limitations which need to be addressed:

Though Java offers very good features for developing distributed applications, it still has certain limitations. Some of them which we encountered are as follows:

Memory Limitations: The Java Virtual Machine easily runs out of memory when dealing with large numbers of images. This prevents our application from providing animations involving large number of images. It can be limiting to other similar applications.

Security: The lack of ability of Java ORBlets to be able to connect to servers running on machines

others than the one from where the ORBlet was downloaded from is a significant limitation. In our case, it prevents our CORBA servers from running in a distributed environment, even though they are capable of doing so. Although these restrictions were added to Java as a security-feature of applets, they can be quite limiting. Some implementations of CORBA like the Visigenic CORBA provide a Gatekeeper which run on the webserver and accepts request on behalf of servers running on other machines.

Lack of AWT portability: The AWT implementations across platform are not totally consistent. The applets which run properly on the Solaris JVM do not run portably on Microsoft's JVM. It is hoped that such inconsistencies will soon be resolved.

Future Work

The Earth Observing System is a continuing project with the main aim of developing a framework for scientific information management and data exploration . Newer and innovative technologies will continue to be analyzed for their effectiveness to provide better solutions. We have identified flexibility and performance as being two key requirements for developing an interactive and efficient system for scientific data exploration. As part of this project, we have analyzed the use of CORBA as a framework for developing distributed systems. There is no doubt about CORBA being a well-defined, high-level object-oriented framework for developing distributed applications. We have exploited its capabilities to provide a framework for parallel scientific computations in order to improve performance. Though CORBA's performance is quite satisfactory, it remains a fact that it cannot approach the performance level of sockets and that is the price that is being paid for gaining the benefits of a high-level framework. However, future frameworks could probably use an new approach which uses a combination of CORBA and sockets as a communication mechanism. The idea is to retain CORBA as a high-level framework and use it for transfer of control information such as queries. But on the other hand, use sockets for the transfer of actual data across the network. This implies that objects maintain a CORBA based control connection and a new connection for data transfer based on sockets. The concept is similar to that used in File Transfer Protocol which use two connections, viz. control and data connection. The data connection is for the actual file transfer while the control connection is used for transfer of control information. However in FTP, both the connections are sockets based. Designing a system based on this approach may not be easy, but it represents an innovative step for future work.

The use of Java Beans component model for developing exploration tools has been the other main focus. Component technology promises to provide the flexibility that is need in such applications for providing interactive data exploration mechanisms. The Java Beans model is being constantly augmented providing better capabilities which can further improve the quality of the system.

CORBA components and CORBA object-by-value are the two extremely useful specifications which are being finalized by the OMG. The use of these features will help solve some of the design issues discussed earlier in a better manner.

References

1. Arnold Ken and James Gosling, The Java Programming Language, Addison Wesley, 1997

2. Budd Timothy, Object Oriented Programming, Addison Wesley, 1997

3. Thomas Mowbray and William Ruh, Inside Corba: Distributed Object Standards & Applictions, Addison-Wesley, 1997

4. Robert Orfali and Dan Harkey, Client/Server Programming with Java and CORBA, Wiley, 1997

5. Robert Englander, Developing Java Beans, O'Reilly, 1997

6. Gamma, Helm, Johnson, Vlissides, Design Patterns - Elements of Reusable Object-Oriented Soft ware, Addison Wesley, 1994

7. Thomas J. Mowbray and Raphael C. Malveau, CORBA Design Patterns, Wiley, 1997

8. Geoffrey Lewis, Steven Barber and Ellen Siegel, Programming with Java IDL, Wiley, 1998

9. Object Management Group, The Common Object Request Broker: Architecture and Specification, 2.0.

10. Sun Microsystems, The JDBC Database Access API specification 1.10

11. Jim Farley, Java Distributed Computing, O'Reilly, 1998