DOVE: Distributed Objects based scientific Visualization Environment

Lalit Kumar Jain & Mark Abbott
College of Oceanic and Atmospheric Sciences
Oregon State University
Corvallis, OR 97331
{*jain, mark*}*@oce.orst.edu*

Abstract

This paper describes the design and performance of a distributed, multi-tier architecture for scientific data visualization. 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 environment for visualization. The server objects use JDBC, the Java application programming interface to SQL databases, to retrieve data from the database. We discuss the system framework and its components and describe an example application and its performance.

Keywords: Distributed Object Computing, Component model, Multi-tier architecture, Scientific data visualization, CORBA, Java IDL, JavaBeans, JDBC, SQL database.

1 Introduction

Two key requirements of an interactive visualization system for scientific data exploration are flexibility and performance. The purpose of a data exploration system is to enable users to uncover and extract relationships hidden in large data sets [2]. This usually requires flexible data manipulation mechanisms which provide a complete coverage of operations needed by the users. The tools available to users should enable them to form ad hoc groupings of data so as to facilitate understanding of the relationships latent within the data. Performance is the other key requirement. Scientific data visualization systems usually require bulk data transfer and involve intensive computation. Designing a system which meets the response time requirements of an efficient, high-performance system is a challenge. In such settings, recent technological advances in areas of component-based software development and distributed object computing have proved highly beneficial. Component-based software development is a promising technology which is gaining rapid popularity because it naturally extends the object-oriented paradigm emphasizing on modularity and reusability. Components are self-contained reusable elements of software that can be controlled and assembled dynamically to form applications [3]. Components can be customized and interactions between them defined at design or run time providing a flexible framework for application development. On the other hand, the distributed computing paradigm brings in the idea of distributing the computation by harnessing geographically-remote resources. In distributed object computing, objects encapsulate an internal state and make it available through a well-defined interface. Client applications can import an interface, bind to a remote instance of it and issue remote object invocations. Thus, distributed object computing has the ability to improve performance through parallel processing. Blending the flexibility of component-based software development with the power of distributed object computing results in a flexible, efficient distributed computing environment for scientific visualization.

Java is a simple, object-oriented, architecture-neutral, portable and multithreaded programming language [1]. It 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. This is where CORBA comes into picture. CORBA is an emerging standard for distributed object computing sponsored by the OMG [4]. 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. The JavaBeans architecture [10] brings the much needed component development model to Java. JavaBeans is a platform-neutral architecture which extends Java's "Write Once, Run Everywhere" capability to component development. The technology enables developers to create reusable software components which can be visually manipulated and dynamically assembled in a wide variety of application builder tools to form working applications.

2 Distributed, Three-tier Framework for Scientific Visualization

2.1 Example Data Domain

This work is being done as part of the Earth Observing System (EOS) project [6] 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 continuous 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.

2.2 Architectural Design

We have a three-tier framework with the client visualization components in the first tier, the application servers in the middle tier and the database 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 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. 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.

3 DOVE System Components

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 manages these server objects and acts as a registry which the clients can use for accessing a reference to the server objects. It implements the following interfaces:

- *Register Interface:* This interface is used by the server objects to register themselves with the Dispenser. A server object registers its unique Interoperable Object Reference (IOR) with the Dispenser. Multiple instances of the same server type may register with the Dispenser.

- *Authentication 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.

- *Scheduler Interface:* The scheduler 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.

Application Server Objects and Server Proxies

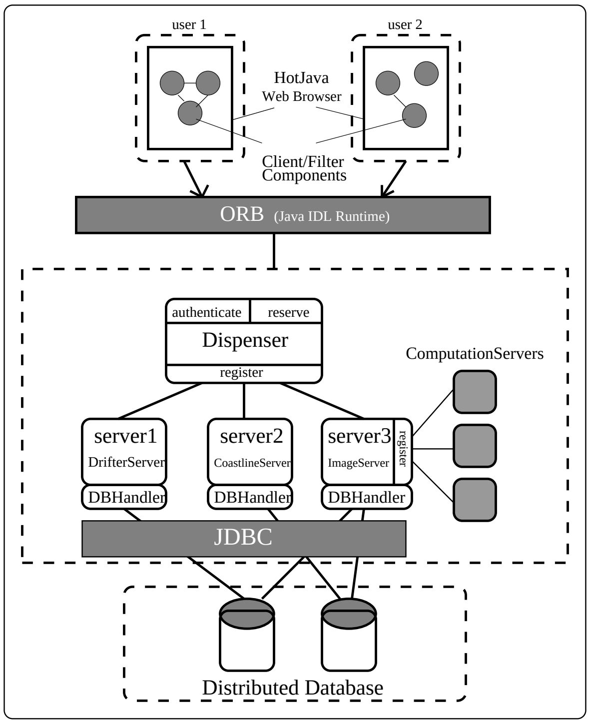
An Application Server Object is a CORBA object. Each Server Object provides uniform access to a portion of the database representing a dataset. A Server Proxy, associated with every Server Object, encapsulates CORBA-specific code required for communicating with the Server Objects. A client component creates an instance of a Server Proxy to communicate with a Server Object. Upon creation, a Server Proxy gets a reference to its Server Object which it will use for further communication with the Server Object.

Database Handlers

Every Server Object uses a Database Handler for communicating with the database. We use JDBC [8] 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.

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 of the task and the number of Computation Server Objects available at runtime. Java's multithreading capability help a Server Objects to invoke multiple Computation Server Objects simultaneously by invoking each of them in a separate thread.



DOVE System Architecture

Figure 1: DOVE System Architecture

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.

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 Component 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.

4 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 needs 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.

4.1 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 [5]. In our system, the Dispenser allows multiple Servers Objects of the same type to register with it.

4.2 Reliability

Distributed application 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 registers 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 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 reference to the Server Object. The Dispenser realises 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 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. The other problem is that the Dispenser itself may crash. In this case, the problem is that the Application Server Objects which have 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.

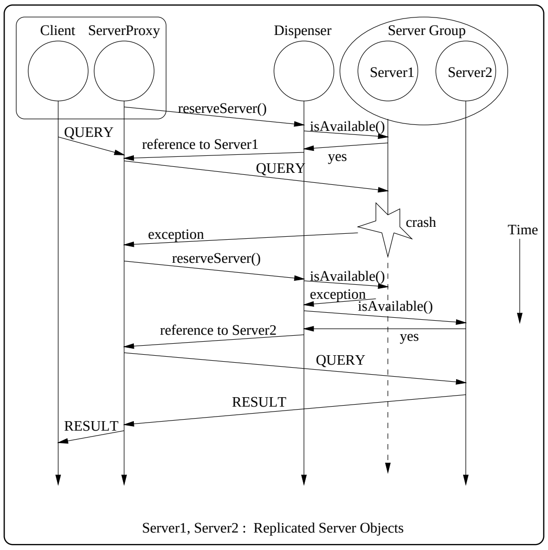


Figure 2: Reliability of the middle tier

4.3 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 [9] needs to communicate with an object B which uses the VisiBroker ORB [7] 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.

5 Example Application and Performance Measurements

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

Application Server Objects:

- *DrifterServer*: This provides access to the drifter data such as the position of a drifter and values measured by it.
- *CoastlineServer*: This provides access to the coastline data corresponding to a given region.
- *ImageServer*: This provides access to the satellite imagery taken by satellites.
- *ImageProcessorServers*: These are the computation servers used by the ImageServers for 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.

Client Components

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

Performance Measurements

We conducted experiments to calculate the speedup achieved by distributed computing in the CORBA framework. 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 ImageProcessorServes. However, in the experiments conducted we measured the total time which include 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 3 shows the performance curve for the test runs.

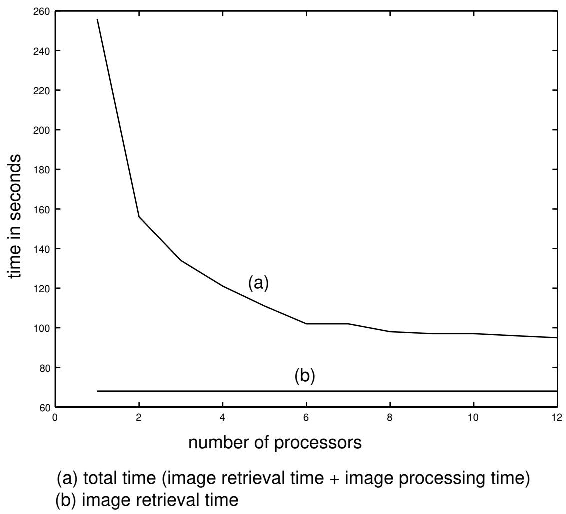


Figure 3: Performance curve: number of processors vs time.

6 Future Work

Java has evolved has a set of APIs for specific purposes. Developing a scientific visualization environment requires dealing with varied fields such as networking, databases, graphics and animation. We have exploited Java's capabilities in all these areas. However the core Java APIs lack a powerful set of classes for image processing. We are awaiting the release of Java Media APIs which will enable us to view our data in different formats such as digital video. We are also looking forward to using the Java 3D API which will provide us with higher level constructs for visualization of 3D data sets. Support for running Java in embedded devices (such as ocean drifters which forms one of the main sources of our data) will add further capabilities such as real-time visualization to the DOVE system.

7 Conclusion

In this paper, we have described a flexible, high-performance framework for scientific data visualization. CORBA server objects running in a distributed environment provide access to database and perform data intensive computation. Clients, implemented as Java Bean components use these server objects for data retrieval and provide data visualization. As computing environments become more distributed, the application development process becomes more challenging. CORBA provides an excellent high-level infrastructure for distributed computing allowing application developers to concentrate on solving the business problems at hand. The JavaBeans component model enables application developement by assembling together reusable software components.

References

[1] James Gosling, Henry McGilton, "The Java Language Environment" , A White Paper, May 1996.

[2] Jade Goldstein, Steven F. Roth, "A Framework for Knowledge-Based Interactive Data Exploration", Carnegie Mellon University.

[3] Robert Englander, "Developing JAVA Beans", The Java Series, O'Reilly, June 1997.

[4] Object Management Group, The Common Object Request Broker: Architecture and Specification, 2.0.

[5] S. Landis, S.Maffeis, "Building Reliable Distributed Systems with CORBA", Theory and Practice of Object Systems, John Wiley Publisher, NY, April 1997.

[6] Earth Observing System project, http://eos.nasa.gov

[7] VisiBroker for Java: Programmer's guide, Visigenic Software

[8] Sun Microsystems, The JDBC Database Access API specification 1.10 via, http://java.sun.com/ products/jdbc/

[9] Sun Microsystems, The Java IDL via http://java.sun.com/ products/jdk/1.2/docs/guide/idl/index.html

[10] Sun Microsystems, The JavaBeans specification