Implementation of Collaborative Scientific Data Visualization

Byeongseob Ki (*bki@top.cis.syr.edu*)
Scott Klasky (*scott@npac.syr.edu*)

Abstract

The area of scientific visualization is one key area where researchers can benifit from collaborative software. We have used Java to develope an infrastructure to perform scientific data analysis. We provide essential tools to allow users to analyze and manipulate their data. We also provide a collaborative framework, where users can exchange data and their own persolnalized filters.

1 Introduction

Java has been developed to be platform-independent and the programming language of the World Wide Web. Just after a year after its inception, Java has become the dominant technology to bring interactive content to the internet. Java contains a programming environment with the Abstract Windowing Toolkit(AWT) and built-in Networking library to make it easy to develope complex applications.

Our approach in designing our system is to provide users with a client-server data analysis syerm by taking full advantage of Java. One of the advantages of using Java to build this application is that it supports peer-to-peer communications, which is vitial for collaboration which is spread across a network. This paper reports on the development of the Java scientific data visualization framework(SciVis).

One of the key features of our system is that filters are *user definable.* A filter is an action which is performed on data set(s) to create new data based on this action. For example, we describe the absolute value function as a filter since this transforms the original data to a new data set which is related to the original data set by the absolute value function. Since our system is collaborative, users can share their filters with collaborators who do not have their filter. We realize that it is impossible to design filters for all applications, so we created API's for users to design their own filters. In the following section, we describe how users can create new filters to help analyze their data.

There are several known applications in Java which have been developed to give applications porting API's [1] [2] [3][TANGO]. Since these application are designed to support wide range of applications, many features can't be optimized and are not suited to some application. Execution speed was a major factor in designing our system, so we choose to develope our own collaborative system. Thus, we have designed collaboration features to fit our application. To support collaboration among users, we used the Remote Method Invocation(RMI) and Object Serialization features of JDK 1.1. Without these, designing complex networked application would have been cumbersome, at best, mainly because we would have to define the precise byte-oriented sequence of message [7]. Section 3 describes the collaboration architecture.

2 Software Components

2.1 Client-Server Model

To get data to our system, we must open a socket connection to the server.

2.2 Data Set

We use a data set which is composed of a series of x,y data points. The data set is composed of multiple time steps.. Each n-dimensional time data has the form of:

time

x_{11} x_{12} \cdots x_{1n} $f(x_{11}, x_{12},$ \cdots $, x_{1n})$

x_{21} x_{22} \cdots x_{2n} $f(x_{21}, x_{22},$ \cdots $, x_{2n})$

\dotfill

x_{m1} x_{m2} \cdots x_{mn} $f(x_{m1}, x_{m2},$ \cdots $, x_{mn})$

Since the data set consits of several time data, we have implemented methods to visualize the data by animating in time. We have provided mechanisms to animate at a certain time rate, proceed to the next time, proceed to the previous time, and to goto a particular time step.

Also, we need the data persistency to back up data objects to a file, or send them to distributed Java servers via RMI. The JDK 1.1 provides us with the Object Serialization mechanism to simplify the problem. As its ma,e probably implies, Object Serialization converts Java Objects into a byte stream, which can then be easily stored to a file. When the file is later read, the byte stream can be easily converted to the original objects [6]. Thus, in our application, we adapted the Object Serialization mechanism to read and data from file or socket.

If the data set is read from the socket, our application will be used as a front end of other application which produces the data set. For example, if one application which is written Fortran 90 or C languages produce the data set, it can write the output data set to the socket which is connected to out server. Then our application will take care of its data analysis and visualization.

2.3 Analyzing the Data

To provide the data analysing tool to the users, we have to survey what features are needed to their purpose. However, it is impossible to include whole analysing features which are needed to every end users into an application. And to write the analysing functions, we need some algorithms for that. These algorithms may be already known or not. Also we should decide which alogorithm is the best. The worse case is that our application programmer should always update the program whenever a better algorithm is found.

So we developed our application with an assumption that the best algorithm and way is to give empty function sets to the users. Therefore, writing the data analysing filters is the user resposibility. The only providing features of our application is giving the easy way for the users to write their own filters to manipulate the data. Then our application will add the user defined filters into the application.

To write the user defined filters, user can overide their filters among *FilterTypeOne*, *FilterTypeTwo*, or *FilterTypeThree* according to needed input. For example, below is the class *FilterTypeOne* in out application,

```
package sv.kernel;

public abstract class FilterTypeOne {

  /**
   * DataSet. The user should manipulate over this DataSet.
   */
  protected DataSet dataSet;

  /**
   * constructor. Nothing to do...
   */
  public FilterTypeOne() {
  }

  /**
   * assign data
```

```
     * @param d DataSet
     */
    public void assignData(DataSet d) {
       dataSet = d;
    }

    /**
     * calculation over DataSet
     */
    public abstract DataSet performFilter();
}
```

In the above class *FilterTypeOne*, the only thing to implement the user defined filter is writing the method *performFilter* over the class *DataSet*. As mentioned earlier, the definition of the class *DataSet* is simple enough to know the structure.

Some filters need their input. For example, think about the filter $x = x + c$. This filter will perform the constant addition over the x values. So this filter need an input c. Currently, we provide three abstract filter classes. *FilterTypeOne* can be used if it needs no input, *FilterTypeTwo* if it needs a constant input like c, and *FilterTypeThree* if it needs a range type input like c_1 - c_2.

So the class *FilterTypeTwo* will be:

```
package sv.kernel;

public abstract class FilterTypeOne {

    /**
     * DataSet. The user should manipulate over this DataSet.
     */
    protected DataSet dataSet;
    /**
     * argument of this class read from user interface
     */
    protected Number val;

    /**
     * constructor. Nothing to do...
     */
    public FilterTypeOne() {
    }

    /**
     * assign data
     * @param d DataSet
     * @param c Number for the argument
     */
    public void assignData(DataSet d, Number c) {
       dataSet = d;
       val = c;
    }

    /**
     * calculation over DataSet
     */
    public abstract DataSet performFilter();
```

```
}
```

If the user defined filters are created by users, it can be easily parts of the application by adding simple interface definition script as follows:

```
<menu> Approximation
  <item>  NAME="Cubic Spline", CLASS="CubicSplineFilter", TYPE=1
</menu>
```

This HTML like script is a definition of the user interface *Popup menu*. There are three fileds in the script item. First, *NAME* field is a name of the filter, and *CLASS* field is a class name which implements the user defined filter. Finally, *TYPE* field is used to let this application know whether it need the user inputs before executing the filter.

3 Collaboration Design

There are several collaborative applications written in Java. But these collaboration applications have their limitation because those are designed to support wide range of applications. So many features can't be optimized and are not suited to some application. Furthermore, they are in the stage to upgrade from JDK 1.0.2 to JDK 1.1 when we write this paper so that they can not use full features of new JDK 1.1.

Currently existing collaborative applications provide their server and APIs to port into their collaboration application. For example, *Collboration Toolsets* developed in Old Dominion University is based on replacement of Java *awt* package. Each *java.awt* component is replaced with exact looking equivalents, but the new component intercepts all the user-events in the application and sends them to the collaboration server. The collaboration server then forwards the events to all other instances of this application, where it is posted to the corresponding component [3]. So to port in *Collboration Toolsets*, the application writer has to override method *handleEvent*.

Another interesting example is *Habanero*, developed in NCSA. *Habanero* is a framework for sharing Java objects with colleagues distributed around the internet. Included are all the networking facilities, routing, arbitration and synchronization mechanisms necessary to accomplish the sharing of state data and key events between collaborator's copies of a software tool [2].

Now, we can think about our application to port to these collaboration tools. But there are some difficulties with this approach. First, these application need predefined methods to collaborate each clients. However filters can't be defined until the end user define their own filters. So the end users also have to know how their filters collaborate with others. Secondly, the filter sets can be different among clients. So if a client perform a filter which is not defined in other client, the application can not be collaborative. So we designed and implemented to resolve these difficulties.

3.1 Architecture

Our application(SciVis) is the Collaborative Scientific Data Visualization tools written entirely in Java using JDK 1.1 's RMI mechanism. RMI was the technology of choice because of its simplicity and inherent portability, and, coupled with the Delegation Event Model, it lets us create an extensible collaboration framework that can be used even with proprietary components. RMI also provides a simple solution to distribute the processing of intensive tasks from the client to the server among different servers [7].

Figure [Ref: arch1] shows the architecture of collaboration. To support collaboratiion among users, we have *Data Repository*, *Visualization Server*, and *Client*.

The *Data Repository* is a data repository to hold the global information of collaboration work. The sturucture of this is shown in the Figure [Ref: Reposit] . The global information are consisted of DataSet, Filter, Connection vectors and Event Queue.

If the user starts the server, then our application will create one *Data Repository* and start the *Visualization Server*. The owner of it's *Data Repository* is this *Visualization Server*. It is connected via RMI between *Data Repository* and *Visualization Server*.

At the time of *Data Repository* creation, the application checks the user defined filters and filled the Filter vector with the user defined filters. Initially DataSet vector and Event Queue are empty.

3.2 Collaboration Work

It is possible that there are several servers on the differnt machines and they can collaborate with each other. This situation are described in the Figure [Ref: migration] .

In this situation, there are two servers(Server A and B) on the different machines. Initially, they have their own *Data Repository*. The owner of Data Repository A is Server A, and the owner of Data Repository B is Server B. This initial state is shown by step 1. Now, Server B wants to join the session to collaborate with Server A. Then Server B sends the request to the Server A via socket. This is step 2. In the step 3, Server A sends the acceptance message to Server B. If the Server A refuses the request from Server B, then nothing happened except that the sign of refusing on Server B. After the acceptance of join, Server B will append its Filters to Data Repository A. This step is shown by step 4. In the step 5, each server that is connected to Data Repository A updates its interface to support newly registered filters.

Now, Server A and B share the Data Repository A, and the owners of Data Repository A are Server A and Server B. So they can access the DataSet, Filter, Connection vector and Event Queue in the Data Repository A. And they are ready to collaboration work.

Figure [Ref: collaboration] shows the collaboration work steps after the initialization. In this figure, Server B executes a filter. Then Server B append the Event identification to the Event Queue of Data Repository A and the the result DataSet to the DataSet vector. After step 1, Server B sends the Event identification to Server A. Then Server A will fetch the new DataSet from the Data Repository, and send the 'done' command to Server B. These steps are described in step 2, 3 and 4. Finally, Server B will remove the Event identification from the Data Repository.

3.3 Deadlock Avoidance

With the steps in the Figure [Ref: collaboration] , event disorder and deadlock can be occurred. To avoid these unexpected situation, we have to add some rules in the appication. First of all, when a Server registers its event to the event queue and if there is an event in the event queue, then this Server should be suspend its execution of filter until first event in the event queue is removed. But its event can be registered.

After the first event of the event queue is removed, next event can be executed. This synchronization is used to avoid event disorder.

To avoid the deadlock, our application allows only creation and removal of the DataSet. So filtered DataSet is always created from the original DataSet. In addition to this restriction, any Server can not leave the collaboration session if there are events in shared event queue. With these schemes, we can avoid the deadlock.

4 Closing Remarks

should be added ...

References

[1] K. Mani Chandy, *Caltech Infospheres Project Overview: Information Infrastructures for Task Forces*, via http://www.infoshperes.caltech.edu/

[2] NCSA, *NCSA Habanero Developer API*, via http://www.ncsa.uiuc.edu/SDG/Software/Habanero/API/index.html

[3] Bjorn J.Kvande, *The Java Collaborator Toolset*, Master's Project Report, Department of Computer Science, Old Dominion University, Aug. 1996. or via http://www.cs.odu.edu/ kvande/Projects/Collaborator

[4] K, Arnold, J. Gosling, *The Java Programming Language*, Addison-Wesley, 1st edition, 1996.

[5] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, Addison-Wesley Developers Press, Sunsoft Java Series, 1996.

[6] Sun Microsystems, *Object Serialization Specification* via http://www.javasoft.com:80/products/jdk/1.1/docs/guide/serialization/index.html

[7] Sun Microsystems, *RMI Specification* via http://www.javasoft.com:80/products/jdk/1.1/docs/guide/rmi/index.html

[8] K. Mani Chandy, E. Schooler, *Designing directories in distributed systems: A systematic framework*, IEEE International Symposium on High Performance Distributed Computing (HPDC-5), Syracuse, NY, Aug 1996.

[9] H. Abdel-Wahab, et. al. *Using Java for Multimedia Collaborative Applications* , Proceedings of the 3rd International Workshop on Protocols for Multimedia Systems (PROMS'96), October 1996, Madrid. or via http://www.cs.odu.edu/ kvande/Projects/Collaborator/

[10] K. Mani Chandy, Adam Rifkin, Paolo A.G. Sivlotti, *A World-Wide Distributed System Using Java and the Internet*, IEEE International Symposium on High Performance Distributed Computing (HPDC-5), Syracuse, NY, Aug 1996.

\centerline [INSERT POSTSCRIPT FIGURE (figure=arch1.ps,height=2in,angle=-90) HERE]

Figure 1 Collaboration Architecture

\centerline [INSERT POSTSCRIPT FIGURE (figure=reposit.ps,height=2in,angle=-90) HERE]

Figure 2 Data Repository Structure

\centerline [INSERT POSTSCRIPT FIGURE (figure=migration.ps,height=2in,angle=-90) HERE]

Figure 3 Joing in the collaboration session

\centerline [INSERT POSTSCRIPT FIGURE (figure=collab.ps,height=2in,angle=-90) HERE]

Figure 4 Performing collaboration work