Abstract

Object Web offers an explosive combination of object/component reusability with the global access - the result could be viewed as an emergent global componentware marketplace but there are still several obstacles on the way. In this thesis I intend to first preview various options that are available today, and review various plausible convergence scenarios. We proceed then to delve deeper into two of these options Java RMI and CORBA. The approach I have used is to compare and contrast the two first for an Image Processing application and then for a Distributed Collaborative Environment (JDCE). The rationale for, and the architecture of, JDCE is explained in detail. In the end we do a quantitative analysis of the response times in case of both RMI and CORBA.
Java Distributed Collaborative Environment (JDCE)- As A Testbed For Distributed Object Technology

by

Shrideep Pallickara

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Engineering in the Graduate School of Syracuse University

August 1998

Approved:______________

Date:__________________

Copyright 1998 Shrideep Pallickara, NorthEast Parallel Architectures Center

All rights reserved
Acknowledgements

The author would like to thank his advisor Dr. Geoffrey C. Fox for the useful inputs he gave. The author would also like to thank Dr. Wojtek Furmanski and the IWT group comprising of Tom Pulikal, Meryem Ispirli among others for the many discussions that resulted in new ideas. The author would also like to thank Daniel Dias at the IBM T. J. Watson Research Center whose inputs on scaling and high-availability went a long way in the overall design of the system.

The JDCE logo was designed by Mahesh Rengaswamy.
# Table Of Contents

**Chapter 1 - Distributed Objects - A Taxonomy** ................................................................. 1
  1.1 Common Object Request Broker ...................................................................................... 1
  1.1.1 IIOP - Internet Inter-ORB Protocol ............................................................................. 2
  1.2 DCOM .............................................................................................................................. 3
  1.3 Java RMI .......................................................................................................................... 4
  1.4 Java-Beans a Component Framework for Java ................................................................. 5
  1.4.1 Component Models & Frameworks ............................................................................. 6
  1.4.2 Introspection ................................................................................................................ 8
  1.4.3 Design Patterns .......................................................................................................... 9
  1.5 DOM – Document Object Model .................................................................................. 11

**Chapter 2 - CORBA, RMI and Java - (Convergence & Chemistry of Object Models)**...... 13
  2.1 Convergence of Object Models ..................................................................................... 13
  2.2 The World of Java and Distributed Objects ................................................................... 14
  2.3 Java CORBA combination ............................................................................................... 16

**Chapter 3 – The Image Processing Application** ............................................................... 18
  3.1 A Simple ImageProcessing Applet .................................................................................. 18
  3.1.1 Loading Images ......................................................................................................... 18
  3.1.2 Preparing a pixel array for Manipulation ................................................................... 20
  3.1.3 Manipulating the Pixel Array .................................................................................... 20
  3.1.4 Creating a Filtered Image ......................................................................................... 21
  3.2 Filter Caffeine .................................................................................................................. 22

**Chapter 4 - Java-CORBA Applications** ............................................................................ 24
  4.1 Defining the IDL ............................................................................................................ 24
  4.2 Generating Client Stubs and Server Skeletons ............................................................... 25
  4.2.1 Anatomy Of the Files Generated ............................................................................. 25
  4.3 Object Implementation Server using BOA (or tie) approach .......................................... 26
  4.3.1 The Object Implementation ..................................................................................... 27
  4.4 Writing a client to access the Object Implementation .................................................. 28
  4.5 Applet- Tags for CORBA Clients .................................................................................. 28
  4.6 Summary of results ....................................................................................................... 29

**Chapter 5 - Java RMI Applications** ................................................................................ 31
  5.1 Defining the Java remote Interface ................................................................................ 32
  5.2 Write the Java remote object server .............................................................................. 33
  5.3 Generating client stub and server skeleton using the rmic compiler ............................... 35
  5.4 The Java applet that remotely invokes the server's method ............................................ 35
  5.5 Summary Of results ....................................................................................................... 37

**Chapter 6 - JDCE - Java Distributed Collaborative Environment** ............................... 40
  6.1 Java & Collaboration API's ............................................................................................ 41
  6.1.1 Specs for Collaboration API's .................................................................................. 41
  6.1.2 Second Generation Java-based Collaboration API's .................................................. 42
  6.2 The Rationale for JDCE .................................................................................................. 45
  6.3 JDCE Features .............................................................................................................. 46
  6.4 Architectural Overview of JDCE ................................................................................... 47
  6.5 Dynamics of Collaboration ........................................................................................... 48
  6.6 Scalability and Fault tolerance Issues .......................................................................... 50
  6.6.1 Scaling: ....................................................................................................................... 51
  6.6.2 Fault tolerance ......................................................................................................... 51

**Chapter 7 - JDCE Widgets** ............................................................................................. 52
  7.1 Session Scheduler ......................................................................................................... 52
  7.2 Data Bahn ...................................................................................................................... 55
  7.3 Client Proxy ................................................................................................................... 59
  7.4 Exception Hierarchy ...................................................................................................... 60
**List Of Illustrative Materials**

| FIGURE 1 - THE OBJECT REQUEST BROKER | ................................................................. | 2 |
| FIGURE 2 - INTER-PROCESS COMMUNICATION USING IIOP | .................................................. | 3 |
| FIGURE 3 - THE RMI SUB-SYSTEM | ........................................................................ | 5 |
| FIGURE 4 - JAVA BEANS IN A NUT SHELL | .............................................................. | 9 |
| FIGURE 5 - DEVELOPMENT PROCESS FOR JAVA-CORBA APPS | .............................................. | 24 |
| FIGURE 6 - DEVELOPMENT PROCESS FOR JAVA RMI APPS | .................................................. | 31 |
| FIGURE 7: REMOTE INVOCATIONS (RMI VS CORBA) | ....................................................... | 38 |
| FIGURE 8 AVERAGE TIMES FOR DIFFERENT ARRAY SIZES | .................................................. | 38 |
| FIGURE 9 HABANERO ARCHITECTURE | ........................................................................ | 43 |
| FIGURE 10 JAVA SHARED DATA ARCHITECTURE | ................................................................... | 44 |
| FIGURE 11 - THE JDCE HIGH LEVEL INTERACTION DIAGRAM | .............................................. | 50 |
| FIGURE 12 - THE JDCE SESSION SCHEDULER | .................................................................. | 54 |
| FIGURE 13 - JDCE DATABAHN MESSAGING PRIMITIVES (BROADCAST & INFORM OTHERS) | .................................. | 56 |
| FIGURE 14 - JDCE DATABAHN MESSAGING PRIMITIVES (WHISPER) | ....................................... | 57 |
| FIGURE 15 - JDCE EXCEPTION HIERARCHY | .................................................................. | 62 |
| FIGURE 16 - THE JDCE EVENT OBJECT | ..................................................................... | 65 |
| FIGURE 17 - THE JDCE MESSAGE FORMAT | .................................................................. | 67 |
| FIGURE 18 - THE JDCE MESSAGE QUEUE IMPLEMENTATION DETAILS | ....................................... | 69 |
| FIGURE 19 - THE JDCE THREADING POLICY | .............................................................. | 70 |
| FIGURE 20 - THREAD POOLING MECHANISM | .................................................................. | 72 |
| FIGURE 21: RESPONSE TIMES - JDCE RMI & CORBA CLIENTS IN COLLABORATION MODE | ....................................... | 94 |

| TABLE 1: CORBA IMAGING RESULTS’ SUMMARY | ................................................................... | 30 |
| TABLE 2: RMI IMAGING RESULTS’ SUMMARY | ................................................................... | 37 |
| TABLE 3: COMPARISON OF JDCE INITIALIZATION PROCESS | .................................................. | 90 |
| TABLE 4: JDCE RESPONSE TIMES FOR 2 DATABAHNS WITH 2 CLIENTS EACH | .................................. | 91 |
| TABLE 5: JDCE RESPONSE TIMES FOR 2 DATABAHNS WITH 4 CLIENTS EACH | .................................. | 91 |
| TABLE 6 JDCE RESPONSE TIMES WITH 2 DATABAHNS WITH 8 CLIENTS EACH | .................................. | 92 |
Preface

The Web has evolved from a static hypermedia system with limited expressiveness into a collection of services provided at different levels and is increasingly being perceived as critical to enterprise computing architectures. The recent onset of distributed object and/or component technologies opens new and interesting avenues for standards based collaboratory infrastructures.

However, selecting a specific direction in the exploding field of distributed object and component technologies is far from being an easy task, illustrated very well both by the myriad options and the uncertainty of overall direction in the field. The questions that we pose ourselves is whether the Web could serve as an infrastructure for both developing and implementing business applications in a, possibly global, distributed and collaborative business environment.

In the approach to answer the question, we adopt the integrative methodology i.e. we setup a multiple-standards based framework in which the best assets of various approaches complement each other. Java Distributed Collaborative Environment as the name suggests is a Distributed Collaborative environment, implemented using both CORBA and RMI. Its primary intent is to serve as a test bed for two of the distributed technology approaches viz. RMI and CORBA. In the end we take examples, which are technology demonstrations, and see how various collaboration issues are dealt with and the advantages of one approach over the other.
Chapter 1 - Distributed Objects – A Taxonomy

Selecting a specific direction in the exploding field of distributed object and component technologies is not an easy task. CORBA offers one promising approach, especially from the large scale computing perspective but there are alternatives such as Java RMI or Microsoft DCOM. Finally, the World-Wide Web Consortium, or W3C, is developing a set of new standards such as XML, DOM, RDF and HTTP-NG which, when combined, can be viewed as yet another, new emergent distributed object model (sometimes referred as WOM). This is likely easier to be adopted by simple to medium complex distributed object/component applications. These alternatives donot themselves guarantee complete and satisfactory software applications. They help users to construct, reuse and connect components, but they supply no guidance for application level semantics or structure – how to create a mosaic of these specific components to solve specific logic.

1.1 Common Object Request Broker

The CORBA standard tries to obviate problems resulting from boundaries such as networks, programming languages, and operating systems. CORBA can be viewed as an environment to support the development of a complete new system or an environment for integrating legacy systems and sub-systems. An implementation of the standard defines a language and platform-independent object bus called an ORB (Object Request Broker), which lets objects transparently make requests to, and
receive responses from, other objects located locally or remotely. Each implementation of the CORBA standard is able to communicate with any other implementation of the standard, the protocol used to achieve this end is the Internet Inter-ORB Protocol (IIOP).

![Figure 1 - The Object Request Broker](image)

1.1.1 IIOP - Internet Inter-ORB Protocol

The IIOP specification defines a set of data formatting rules, called CDR (Common Data Representation), which is tailored to the data types supported in the CORBA Interface Definition Language (IDL). Using the CDR data formatting rules, the IIOP specification also defines a set of message types that support all of the ORB semantics defined in the CORBA core specification. The messaging formats and the CDR constitute the General Inter-ORB Protocol. IIOP is the GIOP message format sent over the Transmission Control Protocol/Internet Protocol (TCP/IP). GIOP messages could be sent over any data transport protocol. Using IIOP any CORBA client can speak to any other CORBA Object. The architecture states that CORBA
objects are location transparent. The implementation, therefore, may be in the same process as the client, in a different process or on a totally different machine.

**Figure 2 - Inter-Process Communication Using IIOP**

1.2 DCOM

Microsoft’s Distributed Computing Platform is based on DCOM, which consists of the Component Object Model (COM) binary standard, augmented with a runtime infrastructure to support component communication across distributed address spaces. DCOM specifies the type and the structure of the interfaces which components must implement. All DCOM components must implement at least one interface IUnknown, which contains basic methods for casting interface references and reference counting. DCOM is itself a relatively simple component standard; its utility resides more in
specialized interfaces such as compound documents, drag-and-drop, and persistent streaming and storage. DCOM’s event notification mechanism is implemented through the IconnectionPoint and several related interfaces. DCOM component interfaces are specified using an Interface Definition Language (IDL) derived from the Open Software Foundation (OSF). DCOM is independent of language and implementation. Until recently, DCOM was restricted primarily to the Windows platform. Maturing ports to Unix and MacOS are reducing DCOM’s platform dependency.

The Microsoft IDE currently supports component development in 3 languages – Visual Basic, Visual C++, and J++ (Microsoft’s Java). The IDE itself is an application built using the native DCOM model, making it extensible and customizable through standard DCOM mechanisms. It also provides mechanisms (add-ins) that help developers customize it to, say, have a different look or enforce desired development patterns.

1.3 Java RMI

This section briefly provides an overview of RMI, and how it would inter-operate with other Distributed Object alternatives like CORBA. Java Remote Method Invocation (RMI) is a set of APIs designed to support remote method invocations on objects across Java virtual machines. RMI directly integrates a distributed object model into the Java language such that it allows developers to build distributed applications in Java. More technically speaking, with RMI, a Java program can make
calls on a remote object once it obtains a reference to the remote object. This can be done either by looking up the remote object in the bootstrap naming service provided by RMI or by receiving the reference as an argument or a return value.

![Image](image.png)

**Figure 3** - The RMI Sub-system

Currently, Java RMI uses a combination of Java Object Serialization and the Java Remote Method Protocol (JRMP) to convert normal-looking method calls into remote method calls. Java RMI supports its own transport protocol today (JRMP) and plans to support other industry standard protocols in the near future, including IIOP. In order to support IIOP as it is today, JavaSoft will define a restricted subset of features of Java RMI that will work with IIOP. Developers writing their applications to this restricted subset will be able to use IIOP as their transport protocol.

### 1.4 Java-Beans a Component Framework for Java

Java is more than a language- its a framework that comprises many components. It includes picoJava, a hardware implementation of the Virtual Machine; the JavaOS, an
operating system implementation, and many application programming interfaces to facilitate development of a broad range of applications. Development of reusable component architectures that will allow large software systems to be designed by combining components from possibly different sources is the intended goal of JavaBeans. JavaBeans provides mechanisms to define components in Java and specify interactions amongst them. Java Beans components provide support wherein component assemblers discover properties about components. One of the other interesting aspect about JavaBeans is that it also accommodates other component architectures such as OpenDoc, ActiveX and LiveConnect. So by writing to JavaBeans the developer is assured that the components can be used in these and other component architectures.

1.4.1 Component Models & Frameworks

A component is a software object that has the following elements

- Properties (public state information readable and/or writeable by a program)
- Methods (Invocable at run-time by a program)
- Events (notifications to a program in response to state changes in the object)

Component Framework:

A component Framework is the design of a set of API's that allow software developers to define a software components that can be dynamically combined together to create an application. A component Framework consists of two major elements - components and containers.
Components encapsulate semantically meaningful application functions. Components differ from other types of reusable software modules in that they can be modified at design time as binary executables instead of a modification at the source code level. Components can have a visual appearance such as a button and can be non-visual such as a data feed monitoring component.

Containers are used to hold an assembly or related components. Containers provide the shared context for components to be arranged and interact with one another. Containers also offer common access to system-level services for a component’s embedded components viz. process threads and memory resources. Containers are sometimes referred to as forms, pages frames or shells, Containers can also be used as components i.e. a container can be used as a component inside another container. Event-based protocols are used to establish relationships between a component and its container.

Since it's a "component architecture" for Java, Beans can be used in graphical programming environments, like Borland's JBuilder, or IBM's VisualAge for Java. This means that someone can use a graphical tool to connect a lot of beans together and make an application, without actually writing any Java code -- in fact, without doing any programming at all. Graphical development environments let you configure components by specifying aspects of their visual appearance (like the color or label of a button) in addition to the interactions between components (what happens when you click on a button or select a menu item).

One important aspect of Java Beans is that components don't have to be visible.
This sounds like a minor distinction, but it's very important: the invisible parts of
an application are the parts that do the work. So, for example, in addition to
manipulating graphical widgets, like checkboxes and menus, Beans allows you to
develop and manipulate components that do database access, perform
computations, and so on. You can build entire applications by connecting pre-
built components, without writing any code.

A "bean" is just a Java class with additional descriptive information. The
descriptive information is similar to the concept of an OLE type library, though a
bean is usually self-describing. Any Java class with public methods can be
considered to be a bean, but a bean typically has properties and events as well as
methods.

1.4.2 Introspection

Because of Java's late binding features, a Java .class file contains the class's
symbol information and method signatures, and can be scanned by a development
tool to gather information about the bean. This is commonly referred to as
"introspection" and is usually done by applying heuristics to the names of public
methods in a Java class.

For those who are queasy about the idea of enforced naming conventions,
JavaBeans provides an alternate approach. Explicit information about a class can
be provided using the BeanInfo class. The programmer sets individual properties,
events, methods using a Bean Info class and several descriptor class types (viz.
Property Descriptor, for specifying properties or the Method Descriptor for specifying methods). To some extent, naming conventions do come into play here as well, as when defining the a BeanInfo class. When an RAD Tool wants to find out about a JavaBean, it asks with the Introspector class by name, and if the matching BeanInfo is found the tool uses the names of the properties, events and methods defined inside that pre-packages class. If not the default is to use the reflection process to investigate what methods exist inside a particular JavaBean class.

1.4.3 Design Patterns

The Beans specification refers to these heuristics of introspection and reflection as "design patterns".

![Figure 4 - Java Beans In a Nut Shell](image)

1.4.3.1 Property

The property metaphor in Java essentially standardizes what is common practice
both in Java and other object-oriented languages. Properties are set of methods that follow special naming conventions. In the case of read/write properties, the convention is that if the property name is XYZ, then the class has the methods setXYZ and getXYZ respectively. The return type of the getter method must match the single argument to the setter method. Read-only or write-only properties have only one of these methods.

1.4.3.2 Boolean properties

This set of properties can have a getter method of the form isXYZ in addition to or instead of getXYZ.

1.4.3.3 Indexed Properties

These set of properties are also supported in Beans, which set or get an indexed value. Indexed properties take an additional integer parameter in their getter and setter methods.

In addition to single and multi-value properties, Java beans defines bound and constrained property types. Bound properties use Java events to notify other components of a property value change; constrained properties let these components veto a change. Constrained properties provide a uniform language-based approach basic validation of
business rules.

Whereas DCOM is language-neutral but platform-dependent, JavaBeans is platform-neutral but language-dependent. DCOM capabilities are built up by composing more elementary binary components. In contrast, JavaBeans component capabilities are implemented as a set of language extensions to the Java Class library. Thus JavaBeans is a set of specialized Java programming language interfaces.

1.5 DOM – Document Object Model

Attempts to use the Web as a distributed computing platform are impeded by the expressiveness of the HTML document standard. An adhoc response from W3C is the incorporation of the <object> tag in HTML 4.0. W3C is also sponsoring standards that could lead to convergence of the component and Web document architectures. Extensible Markup Language (XML), is a meta-model for structured document exchange based on the Standard Markup Language (SGML, an ISO standard). HTML is restrictive in the sense that Web documents are constrained to a set of tags for specifying content and format. XML, on the other hand, supports the definition of customized markup languages. The Resource Description Format (RDF) is a meta-model, to be expressed using XML, for capturing metadata about Web resources. Such metadata could be used by search engines, automated agents, and other Web client and server applications.

The Document Object Model, or DOM, specifies automation interfaces through which scripts or applications can access and manipulate Web documents. Given suitable XML
tags, DOM would allow Web documents to be manipulated as components or containers.

These specifications represent an attempt to define object semantics for distributed
documents.
Chapter 2 - CORBA, RMI and Java -(Convergence & Chemistry of Object Models)

The combination of Java's language features viz. platform-independence, strong security model etc and CORBA's well conceived approaches to static and dynamic, synchronous and asynchronous interfaces, and the rich and comprehensive set of CORBA Services truly provide the right building blocks for a distributed computing paradigm. Thus applets, either pure clients or acting themselves as CORBA servers, can be created and compiled once, and be guaranteed to run in all the environments. With the onset of ORBlets, dynamically downloadable or browser-resident as in the case of Netscape/Visigenic ORB, CORBA boasts an even tighter integration with Java towards the evolving computing paradigm often referred to as the ObjectWeb.

You can use Java to create ActiveX controls or CORBA objects. You can use IIOP to communicate between ActiveX controls or JavaBeans.

2.1 Convergence of Object Models

Before we proceed further it should also be noted that Java Object technology is cross-platform whereas CORBA is cross-language. The similarities in the object models stem from the support for the notion of abstract interfaces distinct from implementations or classes. While having mostly identical Interface inheritance mechanisms, both Java(1.1) and CORBA support the notion of asynchronous notification of events.

This is expressed in Java via the registration/call-back model. In this
model different objects communicate indirectly with one or more listener objects acting as a bridge between the source and sink of data.

In CORBA the client can create a callback object and hand over the object reference to the server. The server can then call the client back when the event occurs. The client thus does not busy-wait for a response from the server.

2.2 The World of Java and Distributed Objects

One of the key design requirements in the Distributed Object environment is to create nuggets of behavior which can be shipped from place to place. Oddly enough, classes provide a nice encapsulation boundary for defining what one of these nuggets are. Java with its mobile code facility (the ability to transfer executable binaries to wherever they need to be executed) handles inter-object communications within its native environment. The Java RMI and Java Serialization interfaces allow Java-Objects to migrate around the network, heedless of what hardware platform they land on, and then control each other remotely. In the world of Web-based Distributed applications Java's multi-threading feature facilitates the optimal framework for the Clients to have richer presentations and the Object-Implementation Servers to support more clients.

- In particular a client can create a thread and have it make a remote operation call, rather than making that remote call directly. In the same framework, a multi-threaded client can receive incoming operation requests (for example, a server call-back) to its objects without having
to poll for communication events.

- In case of Servers too, remote calls can be made without blocking the server. This can also be done within the code that implements an operation or attribute so that some complex algorithm may be parallelized.

The Java distributed object model is similar to the Java object model in the following ways:

A reference to a remote object can be passed as an argument or returned as a result in any method invocation (local or remote). A remote object can be cast to any of the set of remote interfaces supported by the implementation using the built-in Java syntax for casting. The built-in Java instanceof operator can be used to test the remote interfaces supported by a remote object.

The Java distributed object model differs from the Java object model in these ways:

Clients of remote objects interact with remote interfaces, never with the implementation classes of those interfaces. Non-remote arguments to, and results from, a remote method invocation are passed by copy rather than by reference. This is because references to objects are only useful within a single virtual machine. A remote object is passed by reference, not by
copying the actual remote implementation. The semantics of some of the methods defined by class Object are specialized for remote objects. Since the failure modes of invoking remote objects are inherently more complicated than the failure modes of invoking local objects, clients must deal with additional exceptions that can occur during a remote method invocation.

### 2.3 Java CORBA combination

The Marriage of Java-CORBA produces a platform-independent and cross-language model for Distributed Objects. Only the Java Programming language has an inverse mapping to IDL, this allows the programmer to stay in the Java world during the developmental stage of distributed applications. Because of their low-complexity and footprint, Java CORBA implementations can run on thin networks computers and low-end consumer devices. Also Java ORBs allow Java clients to be written without requiring any special software to be installed on the client, thanks to the mobile bytecodes.

Java-CORBA greatly simplifies software upgrades of Clients in large distributed systems, this is more so in Java because of the mobile byte codes and in CORBA due to the Dynamic Invocation Interface. To elucidate further Distributed Computing hitherto comprised of many clients and a few central servers, with the Client now being downloaded onto the Desktop courtesy Java-mobile codes all that is needed is an update of this downloadable client. ORBlets which would
employ the CORBA DII (Dynamic Invocation Interface) produce the same effect on the server side.

Also it imperative that to operate in today's heterogeneous computing environments, distributed applications must work on a plethora of hardware & software platforms. Suitability to business class applications calls for capabilities beyond conventional web based computing - scalability, high availability, performance and data integrity. This is where Java & CORBA play a role which mutually complements each other, Java provides for easier distribution of CORBA-based applications with CORBA providing the where withal of a distributed infrastructure.

CORBA Beans, much like their Java counterparts JavaBeans, will be lightweight, reusable, graphically based components that will give CORBA its long-awaited distributed object model. CORBA Beans will provide mapping from Java to C++ objects and hide Interface Definition Language, making CORBA programming significantly easier.

These factors substantiate the prognosis that the CORBA/IIOP-Java combination would assume a pivotal role in shaping the Internet during the next phase of its evolution.
Chapter 3 – The Image Processing Application

Pretty must be in the eyes of the beholder, but twiddling with images is eminently compelling. As a slight deviation from twiddling with the thumbs, let's get a dab hand at some intricate Image manipulations. Using the Java API for Image Processing, Image objects can be created from raw data. The raw data of an existing Image Object can then be examined, and filters created to have modified versions. These Image Objects can be used in exactly the same way as Image objects created by the Java run-time system, they can be drawn to a display surface, or the result of a filtering operation can be used as input for further image processing. Image data is used by object’s which implement the Image Consumer Interface. The Image Consumer & Image Producer are the basis for Image Processing using the AWT.

3.1 A Simple Image Processing Applet

Support for using images is spread across the java.applet, java.awt, and java.awt.image packages. Every image is represented by a java.awt.Image object. In addition to the Image class, the java.awt package provides other basic image support, such as the Graphics drawImage() methods, the Toolkit getImage() methods, and the MediaTracker class. In java.applet, the Applet class provides an easy way for applets to load images using URLs.

3.1.1 Loading Images

The AWT makes it easy to load images in either of two formats: GIF and JPEG. In the case that class loading the image is an applet sub-class, the following method could be used Image frogImage = getImage(getCodeBase(), "frogImage.gif"). However Java
would not finish loading the image immediately after the getImage function gets called. Thus one could actually load images in separate threads as the program executes.

One can track image loading status either by using the `MediaTracker` class or by implementing the `imageUpdate()` method defined by the ImageObserver interface.

Besides, we also have to wait for all the pixels to be loaded before we can manipulate the images. The following set of statements track the loading of Images.

```java
MediaTracker tracker = new MediaTracker(this);
frogImage = getImage(getCodeBase(), "frogImage.gif");
tracker.addImage(frogImage, 0);
try {
    tracker.waitForID(0);
} catch (InterruptedException e);
```

The code snippet detailed above loads the requested image, “frogImage.gif”, and makes sure that all the pixels are received before continuing. The point to be noted is the fact that we could add several images to the same MediaTracker using `addImage` and then making sure that all images with the same ID are loaded with `waitForID`.

It's easy to display an image using the Graphics object that's passed into the `update()` or `paint()` method. Displaying an Image simply requires invocation of the `drawImage()` method on the Graphics object

```java
g.drawImage(filteredfrogImage, imageWidth, 0, this);
```

The next step involves preparing for manipulation of the Image. This is split up into 3 simple steps viz. Preparing a pixel array for manipulation, manipulating the pixel array, and creating a new image from the modified pixel array and painting it on a display area.
3.1.2 Preparing a pixel array for Manipulation

Create a pixel array whose length equals the product of the width and height of Image, the array of pixels representing the Image can be loaded into the array defined earlier by using the PixelGrabber class. This array has to be one-dimensional, even though it seems to be easier having a two dimensional array and accessing individual pixels by passing x and y coordinates to a two dimensional array. But once we know the width of the image, we can easily calculate the pixels correct offset from its x and y coordinates like this:

\[ \text{Pixel}[y \times \text{imageWidth} + x] = \text{newColor}; \]

Since we would mostly be accessing the pixels in the same order as they were represented in the array though, the one-dimensional way of storing usually suffices in most cases.

```java
int pixels[] = new int[imageWidth * imageHeight];
PixelGrabber pg = new PixelGrabber(frogImage, 0, 0, imageWidth, imageHeight,
    pixels, 0, imageWidth);
try {
    pg.grabPixels();
} catch (InterruptedException e);
```

Once this code snippet executes, we have access to all the pixels in the Image and they are stored in the integer array `pixels`. Now that this step is over and done with we are now ready to move on to some manipulations or filtering.

3.1.3 Manipulating the Pixel Array

The manipulation can be performed on the entire pixel array or on a part of it. To
keep things simple we perform a very simple operation on every pixel in the pixel array. The code snippet which performs this is given below

```java
for (int index=0;index < imageWidth*imageHeight;index++)
    pixels[index]=pixels[index]^0xffffff;
```

A word of caution, since filters normally deal with thousands of pixels at a time it is imperative to thoroughly optimize the inner loop.

### 3.1.4 Creating a Filtered Image

Now that we have the pixel array transformed by the operation we performed, we have one more step to go before we can display the image. This step is the creation of the Image from the manipulated pixel array. This is done as follows

```java
filteredfrogImage=createImage(new MemoryImageSource(imageWidth, imageHeight, pixels, 0, imageWidth));
g.drawImage(filteredfrogImage, imageWidth, 0, this);
```

Java then translates the true-color representation of the image to a format that is suitable for display. This Image is then passed onto the drawImage() method which as explained before paints it onto the screen.

The code which details these ImageProcessing operations is given below.

```java
import java.awt.*;
import java.awt.image.*;

public class filterimage extends java.applet.Applet {
    Image frogImage, filteredfrogImage;
    int imageWidth, imageHeight;

    public void init() {
        MediaTracker tracker = new MediaTracker(this);
        frogImage=getImage(getCodeBase(), "frogImage.gif");
        tracker.addImage(frogImage, 0);
        try {
            tracker.waitForID(0);
        } catch (InterruptedException e) {
        }
        imageWidth=frogImage.getWidth(this);
        imageHeight=frogImage.getHeight(this);
```
int pixels[] = new int[imageWidth*imageHeight];
PixelGrabber pg = new PixelGrabber(frogImage, 0, 0,
imageWidth, imageHeight, pixels, 0, imageWidth);
try {
   pg.grabPixels();
   catch (InterruptedException e);
   for (int index=0;index < imageWidth*imageHeight;index++)
   pixels[index]=pixels[index]^0xffffff;
   filteredfrogImage=createImage(new MemoryImageSource(imageWidth, imageHeight,
pixels, 0, imageWidth));
}
public void paint(Graphics g){
g.drawImage(frogImage, 0, 0, this);
g.drawImage(filteredfrogImage, imageWidth, 0, this);
}

3.2 Filter Caffiene

Filter Caffiene is a library of ImageProcessing routine developed by me at the NorthEast
Parallel Architectures Center. Among the various filters available to the developer are
edge-detection, embossing, hue, stauration, luminosity, luminance. Chromacity, convex,
concave etc. In all there are 48 ImageProcessing filters which perform most of the basic
ImageProcessing operations. All these filters operate on the raw data, and have one
method filter which takes as arguments - an array of pixels, the height and the width of
the Image to be manipulated. I would now detail how you could perform
ImageProcessing using these filters. As an example here I demonstrate a simple inversion
filter. The function of this filter is to simply invert the image, for sake of clarity I merely
include the filter function of the ImgFilter base class, which all filters in FilterCaffiene
implement.

import java.awt.*;
import java.awt.image.*;

public class flipFilter extends ImgFilter {
   public int[] filter(int[] npixel, int nimgw, int nimgh) {

for (int i=0; i< nimgw; i++) {
    for (int j= 0; j < nimgh/2 ; j++) {
        int Temp = npixel[(j*nimgw) + i];
        npixel[(j*nimgw) + i] = npixel[(nimgh-j-1)*nimgw + i];
        npixel[(nimgh-j-1)*nimgw + i ] = Temp;
    }
}
return npixel;
}

As can be seen the filter function performs the necessary manipulation of the array of pixel and returns a new array of pixels.

public void flipImage() {
    flipFilter flip;
    flip = new flipFilter();
    npixel= flip.filter(npixel, nimgw,nimgh);
    nimgh= flip.getHeight();
    nimgw= flip.getWidth();
    updateCanvas();
}

Most of the filters are very easy to use and can be easily plugged into any application. Of course it is assumed that the developer would be responsible for providing pixelsto the filters. This can be done as described in section 3.1 earlier. Also Filter Caffiene returns the manipulated pixels and not the filtered image

createlImage(new MemoryImageSource(imageWidth, imageHeight, pixels, 0, imageWidth));
g.drawImage(filteredImage, imageWidth, 0, this);

The code snippet detailed above can then be used to generate the filtered Image and drawing the result back onto the screen.
Chapter 4 - Java-CORBA Applications

The Development Process for writing a CORBA based application comprises of the following steps. These steps that have been outlined are typical of most CORBA applications. But there could be deviations from this general rule. The application that use to demonstrate building Java-CORBA solutions is the Imaging application introduced in the previous chapter.

1. Define IDL
2. Generating client stub and server skeleton using IDL compiler
3. Object implementation (server) using BOA or TIE approach
4. Client implementation
5. Register Server with an ORB daemon
6. Run client by contacting an ORB daemon

![Building Java-CORBA Applications](image)

**Figure 5** - Development Process for Java-CORBA Apps

### 4.1 Defining the IDL

IDL is a specialized language for defining interfaces, and this is what facilitates the notion of interacting objects, so central to CORBA. The IDL is the means by
which objects tell their potential clients what operations are available and how they should be invoked. The IDL definition defines types of objects, their attributes, the methods they export and the method parameters. We define here the IDL interface for the ImageFilter which we plan to have as a remote object. The ImageFilter performs one function filter which takes an array of integeres (a sequence of long in IDL) and two integers as arguments, and returns an array of integers as results of the filter operation.

```java
interface ImageFilter {
    typedef sequence sequence_of_long;
    ::ImageFilter::sequence_of_long filter(
        in ::ImageFilter::sequence_of_long npixel,
        in long imgw,
        in long imgh
    );
};
```

### 4.2 Generating Client Stubs and Server Skeletons

This is usually done by using the idl2java pre-processor. This automatic generation of stubs frees the developer from having to write those stubs, and frees them from the dependencies of a particular ORB implementation. Compile all the java files which have been generated as a result of the pre-processor operation. Use the –d operation to specify the directory for the compiled classes.

```bash
>idl2java filter.idl

>javac –d classes *.java
```

#### 4.2.1 Anatomy Of the Files Generated

- **ImageFilter.java**: It maps the filter.idl to the corresponding Java Interface.
- **ImageFilterHelper.java**: This provides the bind method, which is used by
clients to locate objects of the ImageFilter class. This also contains mappings for IDL \texttt{out} parameters. (Java natively supports only \texttt{in} parameters).

- \texttt{ImageFilterHolder.java} :: Support for out and inout parameter passing modes requires the use of additional "holder" classes. These classes are available for all of the basic IDL datatypes in the org.omg.CORBA package and are generated for all named user defined types except those defined by typedefs.

- \texttt{ImageFilterOperations.java} :: This has a list of all the operations that the ImageFilter performs.

- \texttt{_ImageFilterImplBase.java} :: Class which Java implementations extend for actual Object implementations.

- \texttt{_sk_ImageFilter.java} :: This implements the CORBA Server-side skeleton for ImageFilter. It unmarshalls the arguments for the ImageFilter Object. In addition this is what is responsible for the convergence of CORBA and Java Object models.

- \texttt{_st_ImageFilter.java} :: This implements the client-side stub for the ImageFilter Object. This is the class which provides for marshalling of functions.

- \texttt{_tie_ImageFilter.java}

\section*{4.3 Object Implementation Server using BOA (or tie) approach}

The ORB uses this information to locate active objects or to request activation of objects on a particular server. This server is primarily responsible for

- Initializing the ORB
- Initialize the BOA
- Create a new instance of the ObjectImpl
- Export the newly created object to the ORB
- Wait for incoming requests

\begin{verbatim}
// CoordinatorServer.java

class FilterServer
{
 static public void main(String[] args)
 {


d
\end{verbatim}
try
{
    // Initialize the ORB.
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init();

    // Initialize the BOA.
    org.omg.CORBA.BOA boa = orb.BOA_init();

    // Create the Bathglass Filter object.
    BathglassFilterImpl _Bathglass =
        new BathglassFilterImpl("Bathglass");
    // Export the newly created object.
    boa.obj_is_ready(_Bathglass);

    System.out.println("Exported the Object Implementation " + _Bathglass);
    // Ready to service requests.
    boa.impl_is_ready();
}
catch(org.omg.CORBA.SystemException e)
{
    System.err.println(e);
}
}

4.3.1 The Object Implementation

This is the class that inherits the functionality of both the CORBA and Java object models. The task is basically to implement the ImageFilter Interface. This calls for defining the functionality for the filter method in the ImageFilter Interface.

The operation performed on the pixel array in the given implementation is the Bathglass Filter.

import java.awt.*;
import java.applet.*;

public class BathglassFilterImpl extends _ImageFilterImplBase {
    /** Construct a persistently named object. */
    public BathglassFilterImpl(java.lang.String name) {
        super(name);
    }
    /** Construct a transient object. */
    public BathglassFilterImpl() {
        super();
    }

    public int[] filter(int[] npixel, int nimgw, int nimgh) {
        System.out.println("Filterng begins on the server-side....");
        int pixell[] = new int[nimgw * nimgh];
int wpixel[] = new int[nimgw];
for (int j=0; j < nimgh ; j++) {
    for (int i=0; i < nimgw ; i++) {
        int a = 10 - i%10 + i;
        a += nimgw*j;
        if (a < nimgw * nimgh) wpixel[i] = npixel[a];
        else wpixel[i] = npixel[nimgw*nimgh -1];
    }
    System.arraycopy(wpixel,0, pixel, j*nimgw, nimgw);
}
return pixel;

4.4 Writing a client to access the Object Implementation

For ease of understanding I have split the process of invoking a remote operation into two methods. As can be seen the connect method simply initializes the Object Request Broker. The Bathglass method perform the action remote method invocation on the ImageFilter Object Implementation.

public void connect() {
    System.out.println("Binding to the Request Broker");
    // Initialize the ORB.
    long startTime = System.currentTimeMillis();
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init();
    long stopTime = System.currentTimeMillis();
    System.out.println("Avg Ping to bind = " + ((stopTime - startTime)/1000f) + " msecs");
    System.out.println("Ready");
}

public void bathglass() {
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init();
    Bathglass = ImageFilterHelper.bind(orb,"Bathglass");
    System.out.println("Bathglass started");
    long startTime = System.currentTimeMillis();
    npixels = Bathglass.filter(npixels,nimgw,nimgh);
    long stopTime = System.currentTimeMillis();
    System.out.println("Avg Ping to complete operation = " + ((stopTime - startTime)/1000f) + " msecs");
    System.out.println("Contrast completed");
    updateCanvas();
}

4.5 Applet-Tags for CORBA Clients.

• When using Communicator, you have two choices on how to program your applets. You may use the ORB embedded in Communicator (which matches VisiBroker for Java 2.5), or you may download the VisiBroker for Java 3.0 ORB on top of the existing version in Communicator. You should download the
VisiBroker for Java 3.0 ORB if you are using features new to VisiBroker 3.0 such as interceptors, event handlers, or SSL.

- To use the ORB runtime that comes with Communicator, you must run your servers and the GateKeeper in backward compatibility mode and add the following param tag to your applet tag in your HTML file.

  `<param name=USE_ORB_LOCATOR value=true>`

- To use VisiBroker for Java 3.0 instead of Communicator's version of VisiBroker, add the following param tag to your applet tag in your HTML file. No special flags are required for the GateKeeper or servers.

  `<param name=org.omg.CORBA.ORBClass value=com.visigenic.vbroker.orb.ORB>`

- If the WebServer is running on default port 80 and the gatekeeper is running on port 15000, you would need to add the following tag.

  `<param name=ORBgatekeeperIOR value=http://osprey7.npac.syr.edu:15000/gatekeeper.ior>`

The applet can now be loaded into a browser or appletviewer.

### 4.6 Summary of results

The following table summarizes the results obtained from CORBA Imaging.

<table>
<thead>
<tr>
<th></th>
<th>Same Machine</th>
<th>Different Machines</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time to initialize the Object Request Broker</td>
<td>393 ms</td>
<td>475 ms</td>
</tr>
<tr>
<td>Time to bind to the Bathglass Filter Object Implementation</td>
<td>40 ms</td>
<td>180 ms</td>
</tr>
<tr>
<td>Time to invoke and complete remote operation</td>
<td>1.263 s</td>
<td>1.743 s</td>
</tr>
</tbody>
</table>

**Table 1:** CORBA Imaging results’ summary
Chapter 5 - Java RMI Applications

The Development Process for writing a RMI based applications comprises of the following steps. These steps that have been outlined are typical of most RMI applications. But there could be deviations from this general rule. In this chapter we use the same imaging application that has been explained and employed in the previous two chapters. Doing this enables us to contrast and compare the approaches we need to use in case of a local object solution, a CORBA solution and the soon to be explained RMI solution.

1. Define the Java remote interface
2. Write the Java remote object server
3. Generating client stub and server skeleton using the rmic compiler
4. The Java applet that remotely invokes the server's method.
5. The HTML code for the web page that references the applet.

Figure 6 - Development Process for Java RMI Applications
5.1 Defining the Java remote Interface

Remote method invocations can fail in very different ways from local method invocations, due to network related communication problems and server problems. To indicate that it is a remote object, an object implements a remote interface, which has the following characteristics:

- The remote interface must be public. Otherwise, a client will get an error when attempting to load a remote object that implements the remote interface.
- The remote interface extends the interface java.rmi.Remote.
- Each method must declare java.rmi.RemoteException in its throws clause, in addition to any application-specific exceptions.
- Any remote object passed as an argument or return value (either directly or embedded within a local object) must be declared as the remote interface, not the implementation class.

Here is the interface definition for the RMIFilter. The interface contains just one method, \texttt{filter}, which returns an array of integers, to the client desiring the filter operation.

```java
import java.rmi.*;

public interface RMIFilter extends java.rmi.Remote {
    public int[] filter(int[] pixel, int nimgw, int nimgh) throws java.rmi.RemoteException;
}
```
5.2 Write the Java remote object server

To write a remote object, one write a class that implements one or more remote interfaces. The implementation class needs to:

- Specify the remote interface(s) being implemented, in this case the RMIFilter Interface.

- Define the constructor for the remote object.

- Provide implementations for the methods that can be invoked remotely. A class can define methods not specified in the remote interface, but those methods can only be invoked within the virtual machine running the service and cannot be invoked remotely.

- Create and install a security manager.

- Create one or more instances of a remote object.

- Register at least one of the remote objects with the RMI remote object registry, for bootstrapping purposes.

The implementation class specifies the remote interface(s) it is implementing. Optionally, it can indicate the remote server that it is extending, which in this example is java.rmi.server.UnicastRemoteObject. As can be seen there is a call to the super() method. The super method call invokes the no-arg
The constructor of `java.rmi.server.UnicastRemoteObject` which "exports" the remote object by listening for incoming calls to the remote object on an anonymous port. The constructor must throw `java.rmi.RemoteException`, because RMI's attempt to export a remote object during construction might fail if communication resources are not available.

For bootstrapping, the RMI system provides a URL-based registry that allows a bind to a URL of the form `rmi://host/objectname` to the remote object, where `objectname` is a simple string name. Once the remote object is registered on the server, clients can look up the object by name, obtain a remote object reference, and then remotely invoke methods on the object. The code which implements this operation is

```java
Naming.rebind("rmi://tarkovsky.npac.syr.edu:6000/rmiFilter", h), assuming of course that the rmiregistry is started on port 6000 on tarkovsky.npac.syr.edu.
```

```java
import java.rmi.*;
import java.rmi.server.*;
import java.net.*;

public class RMIFilterImpl extends UnicastRemoteObject implements RMIFilter {
    public RMIFilterImpl() throws RemoteException {
        super();
    }

    public int[] filter(int[] npixels, int nimgw, int nimgh) throws RemoteException {
        System.out.println("Filtering begins on the server-side....");
        int pixel[] = new int[nimgw * nimgh];
        int wpixel[] = new int[nimgw];
        for (int j=0; j < nimgh; j++) {
            for (int i=0; i < nimgw; i++) {
                int a = 10 - i%10 + i;
                int a = 10 - i%10 + i;
            }
        }
    }
}
```
a += nimgw*j;
if (a < nimgw * nimgh) wpixel[i] = npixel[a];
else wpixel[i] = npixel[nimgw*nimgh -1];
}
System.arraycopy(wpixel,0, pixel, j*nimgw, nimgw);
}
return pixel;

public static void main(String args[]) {

try {
    RMIFilterImpl h = new RMIFilterImpl();
    Naming.rebind("rmi://tarkovsky.npac.syr.edu:6000/rmiFilter", h);
    System.out.println("RMIFilter Server ready.");
}
catch (RemoteException re) {
    System.out.println("Exception in RMIFilterImpl.main: " + re);
}
catch (MalformedURLException e) {
    System.out.println("MalformedURLException in RMIFilterImpl.main: " + e);
}

}

5.3 Generating client stub and server skeleton using the rmic compiler

% rmic -d $RMICLASSES_HOME/ RMIFilterImpl

The -d option indicates the root directory in which to place the compiled stub and skeleton files. The generated stub implements exactly the same set of remote interfaces as the remote object itself. This means that a client can use the Java language's built-in operators for casting and type checking. It also means that Java remote objects support true object-oriented polymorphism.

5.4 The Java applet that remotely invokes the server's method.

The code snippet details the operation that preceedes the remote invocation and the actual remote call. The first operation is to get a reference to "rmiFilter", the
object Reference is then narrowed to the actual remote Interface RMIFilter. This is then followed by a remote invocation of the method in question. These operation are embedded in a try-catch block.

```java
public boolean action(Event event, Object obj) {
    System.out.println("CorbaImaging: action.");
    if ("smooth".equals(obj)) {
        try {
            Remote remoteObject = Naming.lookup("rmi://tarkovsky.npac.syr.edu:6000/rmiFilter");
            /*** Narrow the Object reference to a specific one */
            RMIFilter f = (RMIFilter) remoteObject;

            /*** Make the invocation */
            try {
                pixel = f.filter(pixel, imgw, imgh);
                System.out.println("Filter Operation Complete");
                img = pixelsToI(pixel, imgw, imgh);
                viewer.image = pixelsToI(pixel, imgw, imgh);
                viewer.repaint();
                System.out.println("Image Calculated from Pixels");
                repaint();
            }
            catch (java.rmi.RemoteException exc) {
                System.out.println("Error in Invocation "+ exc.toString());
            }
            }
            catch (Exception exc) {
                System.out.println("Error in lookup");
                repaint();
                return true;
            }
        }
    }
}
```

Once the registry and server are running, the applet can be run. An applet is run by loading its web page into a browser or appletviewer. The HTML File is given below

```html
• <applet code=ImagingClient.class width=400 height=500> </applet>
```
5.5 Summary Of results

The table below (Table 2) summarizes the results obtained from RMI based Imaging.

<table>
<thead>
<tr>
<th></th>
<th>Same Machine</th>
<th>Different Machines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bind to the Bathglass Filter Object implementation</td>
<td>200 ms</td>
<td>614 ms</td>
</tr>
<tr>
<td>Time to invoke and complete Remote operation</td>
<td>246 ms</td>
<td>744 ms</td>
</tr>
</tbody>
</table>

Table 2: RMI Imaging results’ summary

The figure below provides a comparison of the average time to complete the remote operation, for a given number of (successive) remote method calls. As can be seen clearly the RMI operation is approximately 50% faster.
Figure 7: Remote Invocations (RMI vs CORBA)

Figure 8 Average Times for different array sizes
The figure above demonstrates the average time for completion of remote operation for increasing image sizes. The graph is plotted for 3 different cases Local operation, RMI invocations and CORBA based invocation. As can be seen from the figure, as the size of the array increases RMI operations are approximately 50% faster than CORBA based operations.
Chapter 6 - JDCE - Java Distributed Collaborative Environment

Source: Webster's Revised Unabridged Dictionary (1913) [web1913]

1. collaboration \Col'lab'o*ra"tion\, n. The act of working together; united labor [syn: {coaction}]
2. collaboration act of cooperating traitorously with an enemy [syn: {quislingism}]

Etymology tit-bits aside, support for collaboration was one of the main driving forces for the original Web, created by CERN physicists to share data across high energy physics labs. Over the last decade, the Web technologies rapidly evolved and so did the collaboration capabilities of the whole framework. Of course Science and Technology go through fads and fashions much like those of apparel, food and toys, and skeptics are quick to point out flaws which render the web-based collaboration model ineffectual. Like so many approaches that have come and gone in the past, the web-based collaboration model could be here to stay or could fade away like some computational hoola hoop du jour. In the remainder, I intend to explore a few issues concerning collaboration and try to frame answers to the following important questions

a) The Plot - Motives for collaboration

b) The Story so far - 2nd Generation Collaboration Environments

c) The Plot Thickens – Towards Distributed Collaboration Environments

The answers to these questions hopefully would point to where the Web-based Collaboration is headed.
**Definition Of Collaboration**

“Collaboration is a process to reach goals that cannot be achieved acting singly (or, at a minimum, cannot be reached efficiently). As a process, collaboration is a means to an end, not an end in itself. The desired end is more comprehensive and appropriate services that improve outcomes.”

**Simple Guidelines For Collaboration - The first principle:**

Identification Of

- Actions that should be exclusively under the preview of the user
- Actions that would propagate state changes to users working on copies of a particular tool in an established collaborative environment.

**Software Tools & The Need For Collaboration**

- Hitherto, design and development of software tools have focussed on single-user control.
- Independent vendor solutions to address the Collaboration problem, have resulted in inter-operability problems of the software tool.

**6.1 Java & Collaboration API's**

Natural initial focus was on asynchronous collaboratory models (shared databases) which were more recently augmented by the synchronous components (shared displays) and are now further evolving towards the ultimate televirtual (TVR) (shared worlds)
environments. Some essential Web technology thresholds towards TVR included: VRML for 3D interactive front-ends and Java that enabled real-time synchronous connections. It is our belief that if these tools are designed in Java, hardware support would be considerably broadened, expressed albeit by the Java VM.

6.1.1 Specs for Collaboration API's

Collaboration API's need to provide a developer with

Means to retrofit single-user software tools to collaborative tools

Build collaboration Tools from grounds up.

6.1.2 Second Generation Java-based Collaboration API's

6.1.2.1 Habanero from NCSA

- Habanero uses the Presentation-Abstraction-Control model to administer the session and the users visual environment. The abstraction level is the application, the presentation level is responsible for processing the displays and user events while the control part is responsible for the communication aspects between the abstraction and presentation levels. The cross-platform collaboration of different tools is facilitated by sharing Java-objects which implement "The Wrapped Object Model", for routing between the different session-participants. Any class that is used in a Habanero-enabled application must already have been installed on each client's machine. Collaborative objects can share string information and any objects that implement the
Wrapped or Marshallable interfaces. To share objects (other than numbers or strings) developers need to write their own Marshalling and unmarshalling routines, essentially the Marshalling method should snap shot the state of the object (to be shared) in such a way that Unmarshalling can recreate that state in a new object.

Figure 9 Habanero Architecture

The `doEvent()` method in the wrapped object is where the application responds to Collaborative actions. Once an object has been marshalled to other collaborating
participants, the reception of the unmarshalled object is performed in the doEvent method() which takes the Event and Marshallable object as parameters. The doEvent() method overrides both the handleEvent() & action() methods (reserved to process single-user related events) and requires the latter methods to return false for those events which need to be shared. This forces the Java system to pass the event to the Habanero Frame of the application.

6.1.2.2 JSDT (Java Shared Data Toolkit) from Javasoft

![Java Shared Data Architecture](image)

**Figure 10** Java Shared Data Architecture

JSDA provides a Shared Framework for Java at the data level. Data objects are shared over, specific instances of Channels- between two or more Clients. To elucidate further Channels are abstractions for data communication paths in JSDA, whereas Clients are objects which, are the source or destination of data- in a collaboration environment. Any Client object which needs to register its interest, in receiving
messages sent over a channel, must implement the Channel Consumer Interface. On similar lines if a client is interested in being notified about changes in state of some other object it should implement the Channel Observer Interface. It should be noted that a Client which references a Channel could operate on that channel, instinctively its clear that a Client could operate on Multiple channels by referencing multiple channels at the same time.

To register interest in a certain Channel, a Client first needs to join the Session which the Channel is a part of and then the Channel. A Client could be part of multiple Sessions and thus register interest in Channels across those various Sessions. However, a client isn't allowed to have more than one consumer on a channel, though replication of this behavior could be done in a roundabout way. As of now the only data that can be shared are byte Arrays, however its possible to share objects too using the Habanero Model or by creating ByteArrayInput/OutputStreams and using the Object serialization from RMI. Its possible to pass complex data types like images as a sequence of bytes. JSDA also has objects which encapsulate management policies for other given objects.

6.2 The Rationale for JDCE

First generation Collaborative systems were represented by custom Java based collaboratory server technologies such as NCSA Habanero or JavaSoft JSDA, with the former providing for sharing of state and the latter providing a Shared Framework
at the data level. Both these approaches were socket based, thus allowing for real-time communication. However they share the disadvantage of being low-level mechanisms, and requiring the provision of a messaging protocol as well as the message transport. What would really obviate this is a direct object-to-object communication mechanism that would allow objects to call others'

Many fail to understand Web applications for what they are - Distributed Applications. CORBA was developed with the primary aim of building interoperable distributed solutions which work across multiple hardware and software platforms. RMI on the other hand extends Java naturally into the Distributed Object computing domain, introducing local/remote transparency for method invocations on Java objects. The intersection of these solutions catalyze the development of dynamic solutions and opens new interesting avenues for the standards based collaboratory infrastructures. Java Distributed Collaborative Environment is an attempt to explore these concepts and the feasibility (quantified in terms of high availability, scalability and fault tolerance) of such systems. In the sections that follow there is a detailed discussion about the scaffoldings that JDCE is built upon, the retrofitting process and feasibility analysis.

6.3 JDCE Features

- A distributed-approach towards Collaborative Environments.

The environment provides for Collaboration between an RMI-Client and a
CORBA-client

- A Distributed Event and Exception Hierarchy.
- Based on the CORBA Naming Service.

6.4 Architectural Overview of JDCE

JDCE provides for a distributed collaboration environment. The sessionScheduler Object is responsible for the creation and maintainence of the data communication paths –dataBahn’s. The name dataBahn is akin to the autoBahns available for automobiles, dataBahns signify the paths available for communication. There is one dataBahn per group per application session. The dataBahn’s support various messaging primitives such as broadcast, whisper, and send-to-others. The messages sent across the collaborative session have a format specified by the JDCEMessage type. There is support for both basic data types and custom objects to be sent to other participants present in the system.

JDCE events report various events which take place during the Collaborative session. The events encapsulate sufficient information to describe the complete occurrence of the activity. Such occurrences include the joining or expulsion of a client, or an invitation to a client to join the application session. When a client invokes one of the messaging primitives, the messages are first added to a message Queue which allows for priority messages to be added to the message Queue. The prioritized queue is then acted upon by the worker threads. The worker threads, one per-client, consume one message after the other, from the head of the queue, till such time that there are no
messages left in the message queue. For faster addition/removal of messages the message queue is implemented as a Doubly Linked List.

The Worker threads, irrespective of whether they service a RMI Client or a CORBA Client, are scheduled by a thread scheduler. The thread Scheduler schedules various threads after a certain time-slice, based on shifting the priorities of threads. Events or actions which are not authorized by the system result in a distributed exception being thrown. All exceptions in JDCE are checked exceptions thus forcing the application developer to be aware of the failures that could possibly take place as a result of his actions.

6.5 Dynamics of Collaboration

The Client initiates a bind to the sessionScheduler Object. Given that this is successful, in the event that there is a Distributed Directory service and the Active Object server is in place, the Client is now ready to invoke the IDL-defined operations defined for the System. Nevertheless, an invocation of any of these aforementioned operations should be preceded by a successful reception of a remote handle to these objects. For a caller (client, peer, or applet) to be able to invoke a method on a remote object, that caller must first obtain a reference to the remote object. Most of the time, the reference will be obtained as a parameter to, or a return value from, another remote method call.

- It starts with the `createBahn(String PartyName, String`
_applicationServerName) throws jdceBahnException function which would return a true in the event that a new dataBahn Object has been instantiated or a jdceBahnException is raised to signify the prior existence of the desired Bahn. It is the Scheduler's job to signal the appropriate notification to the Clients and perform appropriate house-keeping to reflect new instances of dataBahn's.

• The Client now has the option to decide whether he wishes to join an existing Bahn (party) or initiate the existence of a new one. In the latter case Step[1] is repeated as mentioned earlier. This operation also raises the jdceBahnException to signify the non-existence of the dataBahn. Once the process is over, the Client gets a handle to the dataBahn Object by invoking jdce.byteways.dataBahn getDataBahnHandle(String PartyName, String _applicationServerName) throws jdceBahnException.

• Once Steps I and II are over and done with, the Client is in a Distributed Collaboration mode, and can invoke operations specified in IDL definitions for the dataBahn. These operations are preceded by public int register(String _clientName, clientProxy _clientObjRef) throws jdceBahnException, jdceClientException, the registration process is a pre-requisite to switching to the Collaboration mode. Once the registration process is over and done with the DataBahn serves as a conduit for messages and the Client could invoke
operations such as `public boolean broadcastMessage(jdceMessage _message)`,
`public boolean whisperMessage(jdceMessage _message, String _clientName)`
throws `jdceClientException`, `jdceBahnException` etc.

Figure 11 - The JDCE High Level Interaction Diagram

This is just another demonstration of the complementary roles Java & CORBA play
in distributed environments. Java provides for easier distribution of CORBA-based
applications with CORBA providing the where withal of a distributed infrastructure.
In summary, CORBA offers both a potential candidate for universal wire protocol,
IIOP, and a natural collaboratory framework based on shared CORBA objects and
flexible message filtering mechanisms offered by the Event Service.

6.6 Scalability and Fault tolerance Issues
The ideal system would be perfectly scalable and fail-proof. This, of course, is
impossible pragmatically considering the non-utopian world that we live in. System
developers have finite resources to devote to reliability and consumers would be willing to pay approximately the same for these features. Thus systems need to try and reasonably approximate the scaling/fault-tolerance issues.

**6.6.1 Scaling:**

Specifically this addresses the issue of replication of Servers across different participating hosts in a Collaborative environment, when the load (participating sessions) on a server crosses a certain threshold. The API supports the notion of groups within a certain session, and since the Object-Implementations handling the groups can be hosted on different machines, the problem of Scalability is addressed to a considerable extent.

**6.6.2 Fault tolerance**

This concerns the problem of migrating sessions to a different participating host should the machine hosting the server crash - this with minimal or little disruption.

1. The ObjectServices agent which is a distributed directory service allows for migration of sessions to a different participating host in the case that a session terminates unexpectedly on one of the hosts

2. The Events are persistently stored by the Event Channel to ensure that events are not lost on system failures.
Chapter 7 - JDCE Widgets

In this chapter we take a closer look at the different widgets upon which JDCE is built. First, there is the sessionScheduler, which maintains a registry of the dataBahns, communication paths, that are present in the system. There is then a discussion of the Client Proxy, which is the callback object in both the RMI and CORBA case. I follow this up with an explanation of the message formats, event types and the distributed exception heirarchy, which provide for messaging primitives, notification and authorizations. There is then the rationale for the use of a messageQueue and Worker threads. The threading policy used throughout the system is one-thread-per-client-session. This doesn't preclude the client from being registered to different application sessions or from being registered to two groups within the same application session. The chapter is wrapped up with a discussion of the threadScheduler and the CORBA naming service.

7.1 Session Scheduler

The Session Scheduler provides the basic house-keeping necessary for the JDCE Collaborative Environment. The scheduler maintains a registry of the all the applications and dataBahns that were spawned by the scheduler. Invocation of any of the aforementioned operations should be preceded by the successful receipt of a handle to the sessionScheduler. It should be understood that Clients of remote objects interact with remote interfaces, never with the implementation classes of those interfaces. Nonremote arguments to, and results from, a remote method invocation are passed by copy rather than by reference. As can be seen, each method declares java.rmi.RemoteException in its throws clause, in addition to the jdce exceptions that would be described briefly. All the JDCE exceptions are checked exceptions hence client programmers who
wish to invoke these methods would need to either catch and handle the exception within the body of their methods, or declare the exception in the throws clause of their methods. In the RMI based environment, when a `java.rmi.RemoteException` is thrown during a remote method invocation, the client may have little or no information on the outcome of the call - whether a failure happened before, during, or after the call completed.

```
package jdce.scheduler;
import jdce.util.exception.*;

public interface RMISessionScheduler extends java.rmi.Remote {
    String[] listApplications()
        Throws java.rmi.RemoteException, jdceRMIBahnException;
    String[] listBahns(String _applicationServerName)
        Throws java.rmi.RemoteException, jdceRMIBahnException;

    boolean createBahn(String PartyName ,String _applicationServerName)
        throws java.rmi.RemoteException, jdceRMIBahnException;

    boolean joinBahn(String PartyName ,String _applicationServerName)
        throws java.rmi.RemoteException, jdceRMIBahnException;

    jdce.byteways.RMIDataBahn getDataBahnHandle(String PartyName,String _applicationServerName)
        throws java.rmi.RemoteException, jdceRMIBahnException;

    boolean destroyBahn( String PartyName ,String _applicationServerName)
        throws java.rmi.RemoteException, jdceRMIBahnException;
}
```
Reverting back to the specifics of the session Scheduler, the points which need to be noted is that the scheduler maintains a list of all applications that have been registered with it. This information is then linked with all the DataBahns that have been created by the scheduler. The intersection of these two registries provides a listing of all the data Bahns that have been created for a specific application. This information is passed to the Client as an array of Strings. The client can then decide to either join (or create) a data Bahn. Besides this the scheduler provides operations for getting a remote reference to a dataBahn as also a method to destroy a specific Bahn.
7.2 Data Bahn

DataBahns provide the basic abstraction of a communication channel, which support different message passing primitives besides providing for registration and de-registration from the communication channels. To put it briefly the DataBahn is essentially a conduit for message passing. Invocation of any of the operations listed in the DataBahn Interfaces should be preceded by the register method being called. This registration results in the creation of a worker thread for the client to handle callbacks from the server-side. Besides this the thread is also add to the thread scheduler to reduce the average response time in message processing.

![Diagram of JDCE Message Passing Primitives (Broadcast & InformOthers)]
The figure above demonstrates the details of the BROADCAST and INFORM_OTHERS message primitives. In the former case the message is first sent to the server and the server initiates callbacks on all the clients registered on the DataBahn. In this case the client which initiated the communication also receives a response from the server. In the latter case i.e. INFORM_OTHERS all the clients excluding the one which initiates the communication receives a response from the Server. The two modes are different in the terms of the response time from the Server, in the latter case the client could just perform the actions which would be performed in the case that the message was received from the server and still have the effect of BROADCAST with a much better response time on the Client which initiated the operation. It is up to the client what to do with the message. It may even choose to ignore the message, for example if such a message is unauthorized or out of context. Authorization checks and other security responsibilities are best handled by clients. If a client has buggy code, only that client is affected. A bug in the server affects all clients. The only thing the server may do is the initial check to see if the client has authority to connect to it.

An ideal implementation of group broadcast would use multicast sockets (java.net.MulticastSocket) to communicate efficiently with other group members. However, applets are not allowed to open multicast sockets, and even if they were, operation of a multicast socket through the Internet
requires the underlying support of the Multicast Backbone (MBONE), which is not yet widely deployed. Moreover, multicast sockets only support unreliable datagrams, and would thus require designing a reliable protocol.

Figure 14 - JDCE DataBahn Messaging Primitives (Whisper)

The figure above demonstrates the whisper primitive where a Client sends a message only to a specific client. The Client initiating the whisper gets a response which indicates the success or failure of the operation reflecting the authorizations/presence of the Client or the lack thereof, this is clearly
reflected in the exceptions that would be thrown.

It should be noted that the RMIDataBahn and the CORBA-based DataBahn supports the same set of operations. The DataBahn also provides operations for expelling a certain client, listing the number/information of members registered to the DataBahn. The exceptions thrown in both the RMI/CORBA case are similar but for the additional RemoteException thrown in the RMI mode. The RMIDataBahn Interface detailing all these operation are given below, the IDL definition for the DataBahn has been omitted here for the sake of clarity. The complete IDL definition would nevertheless be available in the appendix.

```java
package jdce.byteways;

import jdce.util.message.*;
import jdce.util.exception.*;

public interface RMIDataBahn extends java.rmi.Remote {

    String getApplicationServerName() throws java.rmi.RemoteException;
    int numberOfMembers() throws java.rmi.RemoteException;
    String[] getClientNames() throws java.rmi.RemoteException;
    boolean isEmpty() throws java.rmi.RemoteException;
    int register (String ClientName,
                jdce.client.RMIClientProxy clientObjRef)
                throws java.rmi.RemoteException,
                jdceRMIBahnException, jdceRMIClientException;
    boolean expelClient(String _sender, String _expelee)
                throws java.rmi.RemoteException,
                jdceRMIBahnException, jdceRMIClientException;
    boolean deregister(String ClientName)
                throws java.rmi.RemoteException,jdceRMIClientException;
    boolean broadcast(String Message) throws java.rmi.RemoteException;
    boolean broadcastMessage(jdce.util.message.jdceMessage Message)
                throws java.rmi.RemoteException, jdceRMIBahnException;
    boolean inform(jdce.util.message.jdceMessage Message)
```
throws java.rmi.RemoteException, jdceRMIBahnException;

boolean whisper(String Message, String clientName)
    throws java.rmi.RemoteException,
           jdceRMIClientException, jdceRMIBahnException;

boolean whisperMessage(jdce.util.message.jdceMessage Message,
                        String clientName)
    throws java.rmi.RemoteException,
           jdceRMIClientException, jdceRMIBahnException;
}

7.3 Client Proxy

The Client Proxy interface defines the appropriate methods which would be called by the DataBahn Object Implementation in response to the events taking place on it. This includes the `dataReceive(java.lang.String Message)` which would be called in case the message sent by any of the registered (on the DataBahn) clients is a String. The `messageReceive(jdce.util.message.jdceMessage Message)` is called in the case the messages are exchanged in the `jdce.util.message.jdceMessage` format, this particular message format is explained in detail in a later sub-section. The `clientEvent(jdce.util.event.jdceEvent evt)` is called in the case that a client has just registered to the DataBahn, or has been deregistered/expelled. In any case the `jdce.util.event.jdceEvent` object created by the DataBahn provides the relevant information.

package jdce.client;

public interface RMIClientProxy extends java.rmi.Remote {

    boolean clientEvent(jdce.util.event.jdceEvent evt)
        throws java.rmi.RemoteException;

    boolean dataReceive(java.lang.String Message)
        throws java.rmi.RemoteException;

    boolean messageReceive(jdce.util.message.jdceMessage Message)
        throws java.rmi.RemoteException;

}
7.4 Exception Hierarchy

Exceptions are the customary way in Java to indicate to a calling method that an abnormal condition has occurred. When a method encounters an abnormal condition (an exception condition) that it can't handle itself, it may throw an exception. Exceptions have several benefits. First, they allow you to separate error handling code from normal code. You can surround the code that you expect to execute 99.9% of the time with a try block, and then place error handling code in catch clauses -- code that you don't expect to get executed often, if ever. This arrangement has the nice benefit of making your "normal" code less cluttered. In general, one should throw an exception and never throw errors. Error, a subclass of Throwable, is intended for drastic problems, such as OutOfMemoryError, which would be reported by the JVM itself. On occasion an error, such as java.awt.AWTError, could be thrown by the Java API. In your code, however, you should restrict yourself to throwing exceptions (subclasses of class Exception).

A checked exception is some subclass of Exception (or Exception itself), excluding class RuntimeException and its subclasses. Unchecked exceptions are RuntimeException and any of its subclasses. Throwable and any subclasses besides the Exception family, including Error and its subclasses, also are unchecked. But as one should be focusing on throwing exceptions only, the
decision should be whether to throw a subclass of RuntimeException (an unchecked exception) or some other subclass of Exception (a checked exception).

If you throw a checked exception (and don't catch it), you will need to declare the exception in your method's throws clause. Client programmers who wish to call the method will then need to either catch and handle the exception within the body of their methods, or declare the exception in the throws clause of their methods. Making an exception checked forces client programmers to deal with the possibility that the exception will be thrown.

- If you throw an unchecked exception, client programmers can decide whether to catch or disregard the exception, just as with checked exceptions. With an unchecked exception, however, the compiler doesn't force client programmers either to catch the exception or declare it in a throws clause. In fact, client programmers may not even know that the exception could be thrown. Either way, client programmers are less likely to think about what they should do in the event of an unchecked exception than they are in the case of an unchecked exception.

The JDCE Exceptions are all checked exceptions and hence client application writers need to be aware of the exceptions that would take place. Though these exceptions makes development of JDCE applications seem rather
clumsy because it forces you to think about something you would rather ignore, the applications are ultimately much more solid and reliable.

![JDCE Exception Hierarchy Diagram]

**Figure 15 - JDCE Exception Hierarchy**

```java
module java {
    module lang {
        extensible struct Object;
    };
};
module jdce {
    module util {
        module exception {
            exception jdceBahnException {
                long type;
            };
            exception jdceClientException {
                long type;
            };
        };
    };
};
```

**IDL Definition**
The code below demonstrates the try-catch block employed by a client application to register itself to the dataBahn. The method `register` belongs to the dataBahn object, and it throws both a `jdceBahnException` and `jdceClientException` if it encounters an error. These errors are detailed shortly below.

```java
try {
    thisClient._chatBahn.register(thisClient.clientName,
                               (jdce.client.clientProxy)control);
    System.out.println("Registration Succeeded");
} catch (jdceBahnException e) {
    System.out.println(e.typeToString(e.getType()));
    System.out.println("Bahn Exception");
} catch (jdceClientException e){
    System.out.println(e.typeToString(e.getType()));
    System.out.println("Client Exception");
}
```

If a Client exception has occurred. The following exception types can occur:

- **NAME_IN_USE** - a client with a similar name exists.

- **ALREADY_BOUND** - The client & the obj Reference have been registered.

- **CLIENT_NOT_REGISTERED** - The registration process failed.

- **NOT_BOUND** : The client has not been bound to the Session Scheduler, Collaboration mode not yet activated for Client.

- **INVALID_OPERATION**: Client not authorized for the particular operation.
• UNKNOWN : Reason unknown to the JDCE Environment.

If a Bahn exception has occurred. The following exception types could occur.

• BAHN_CREATION_FAILED : The Bahn was not created successfully.

• BAHN_EXISTS : The Bahn exists, i.e. there is Bahn for a given application under the same name.

• BAHN_JOIN_FAILED : Join to the dataBahn failed.

• NO_SUCH_BAHN: Reference to a non-existent dataBahn.

• PERMISSION_DENIED: One of the operations on the dataBahn (expel, invite etc ) not completed due to insufficient permissions for the client which invoked the operation.

• UNKNOWN : Reason unknown to the JDCE Environment.

7.5 Client Events - A Reporting mechanism

A software event is a piece of data sent from an object, notifying the recipient object of a possibly interesting occurrence. Basically any occurrence can be modeled as an event and the relevant information regarding the event can be encapsulated within the event. To put it simply an event is self describing. Firing
and responsive handling of events are one of two ways that objects communicate with each other, besides invoking methods on each other. Both these techniques are in a sense one and the same. The client notification mechanism is available only to the valid registered clients on a dataBahn. The reporting mechanism is not available across dataBahns, i.e. a client on a Chat_Bahn cannot receive client activities on a WhiteBoard_Bahn.

```java
module jdce {
    module util {
        module event {
            extensible struct jdceEvent {
                long type;
                string clientName;
            };
        };
    };
}
```

IDL Definition

- `java.lang.Object`
- `jdce.util.event.jdceEvent implements java.io.Serializable`

**Figure 16 - The JDCE Event Object**

There are three types of client changes:

- CLIENT_JOIN - a client has joined the dataBahn.
- CLIENT_LEAVE - a client has left the dataBahn.
- CLIENT_EXPEL - a client has been expelled by the creator of the bahn.
7.6 Messages

The jdceMessage format provides a clear representation of the Message Origins, the applicationSession it is intended for, the priority of the message, the transport protocol it uses to reach the sessionScheduler and the message Content, which could be any **Serializable** Object. Arguments to, or return values from, remote methods can be of any Java type, including objects, as long as those objects implement the interface `java.io.Serializable`. Most of the core Java classes in `java.lang` and `java.util` implement the Serializable interface. The only widget in the Message format which could possibly need a serialization method implemented is the `content`.

- `java.lang.Object`
- `jdce.util.message.jdceMessage` implements `java.io.Serializable`

```java
module java {
    module lang {
        extensible struct Object;
    };
};

module jdce {
    module util {
        module message {
            extensible struct jdceMessage {
                string sender;
                string applicationType;
                string messageType;
                string protocolType;
                ::java::lang::Object content;
            };
        };
    };
};
```

**IDL Definition.**

The Message Object Implementation consists of the following methods, which conform to the *JavaBeans Design Patterns*:

- `GetApplicationType()`
- `getBahnName()`
- `getContent()`
- `getProtocol()`
- `getSender()`
- `setApplicationType(String)`
- `setBahnName(String)`
- `setContent(Object)`
- `setProtocolType(String)`
- `setSender(String)`
7.7 Message Queue

Active message handling requires that the system possess a message queue, which collects messages from other active clients and distributes them in first-in, first-out (FIFO) order to the worker threads responsible for invoking the callbacks. The jdceMessageQueue is implemented as a **Doubly Linked List**. Each Widget in the list is an object with pointers to the next and previous Widgets. The next field of the tail-element and prev field of the head are set to NULL. The queue provides support for Priority messages by adding them to the HEAD of the list instead of appending them to the List.
class QueueWidget {
  Object object;
  QueueWidget prev;
  QueueWidget next;

  QueueWidget(Object object,
               QueueWidget queueWidget1, QueueWidget queueWidget2) {
    this.object = object;
    prev = queueWidget1;
    next = queueWidget2;
  }
}

The Worker threads consume messages which are available on the MessageQueue, while the Clients add messages (priority/otherwise) to the MessageQueue. One of the primary reasons for implementing the Message Queue as a Linked List was the faster addition and removal of messages from the MessageQueue. The messages in the Message queue are never processed out-of-order i.e. the element consumed is always the HEAD of the message Queue, though it must be noted that there is method available to remove a specific element from the List. Besides this the message queue also provides for reporting availability of data on the queue and provides for returning the queue Contents as a stream of bytes or an array of Strings. The queue also has a in-lined class the QueueEnumerator which provides an Enumeration of the QueueContents.

class QueueEnumerator implements Enumeration {
  private QueueWidget head;
  QueueEnumerator(QueueWidget queueWidget) {
    head = queueWidget;
  }
  public boolean hasMoreElements() {
    if (head == null)
      return false;
    else
      return true;
  }
  public Object nextElement() {
    if (head == null)
      return null;
    Object object = head.object;

    // The rest of the code...
  }

7.8 Worker Threads

A thread, or a sequence of instructions being executed in a program, is a lightweight process that reduces overhead by sharing fundamental parts with other threads. Threads are lightweight so that there can be many of them present within a process. Using multiple threads provides concurrency within an application and improves performance. Applications can be structured efficiently with threads servicing several independent computations simultaneously. To maintain uniform sequencing of messages received in the Collaborative system, the thread pooling policy employed in the system is the thread-per-session policy. This ensures that
messages are never handled out of sequencing. Priority messages are handled by
the MessageQueue implementation which can add a priority message to the top of
the messageQueue.

With the thread-per-session policy employed by JDCE, threading is driven by
connections between the client(RMI or CORBA) and server processes. The
thread-per-session policy allocates a new worker thread, every time a new client
registers itself with a dataBahn, to handle the call-backs from the server-side..
(With thread-per-session, there is only a combination thread-the reader/worker
thread. The functionality is not split into two different threads like it is in thread
pooling.) The worker thread handles all the requests received from a particular
client. When the client disconnects from the server, the worker thread is
destroyed.

Figure 19 - The JDCE Threading Policy
The RMIWorker Thread contains references to the corresponding RMIClient while the CorbaWorker contains references to the CorbaClient, thus facilitating callbacks. We now have a brief look at the class hierarchy for the Worker threads.

- class java.lang.Thread (implements java.lang.Runnable)
  - class jdce.util.message.jdceRMIWorker
  - class jdce.util.message.jdceWorker

The run method in both the worker threads check to see if the message at the head of the message queue is intended to be received by the client whose reference it possesses. If it is, in fact, intended for the client, the worker thread performs a server-callback on the client reference. The worker could call any of the following methods on the client – **dataReceive()**, **messageReceive()** and **clientEvent()** based on whether a jdceMessage object is to be received, a String message is to be received and whether the client is to be notified of any occurrence on the dataBahn respectively. The Worker thread for a particular client remains active till such time, that the client has de-registered from the dataBahn. In the event that the client is registered to two different dataBahns, when the client de-registers from one of the bahns, only the worker thread corresponding to the de-registration is removed.
Figure 20 - Thread Pooling Mechanism

Both the RMIWorker and CORBAWorker Threads are scheduled by the \textit{jdceThreadScheduler}. The fact that the ThreadScheduler schedules both the RMI and CORBA threads facilitates collaboration involving RMI and CORBA Clients.

7.9 The CORBA Naming Service

The Naming Service provides the ability to bind a name to an object relative to a naming context. A naming context is an object that contains a set of name bindings in which each name is unique. To resolve a name is to determine the object associated with the name in a given context. Through the use of a very
general model and dealing with names in their structural form, naming service implementations can be application specific or be based on a variety of naming systems currently available on system platforms.

Graphs of naming contexts can be supported in a distributed, federated fashion. The scalable design allows the distributed, heterogeneous implementation and administration of names and name contexts. Because name component attribute values are not assigned or interpreted by the naming service, higher levels of software are not constrained in terms of policies about the use and management of attribute values.

Through the use of a "names library," name manipulation is simplified and names can be made representation-independent thus allowing their representation to evolve without requiring client changes. Application localization is facilitated by name syntax-independence and the provision of a name "kind" attribute.

### 7.10 Summary

Shared tools quickly become important as distributed, often virtual/ephemeral, teams are used in industry and government to pool resources in addressing specific problems. If the principal design tools in use by the scientists and engineers could themselves be shared between the colleagues, a greater degree of efficiency could be affected. By design JDCE provides a convenient framework for both, retrofitting single-user applications or developing collaboration aware applications. JDCE is written (to date) exclusively in Java (TM) from Sun Microsystems. As such, it runs on any platform for which the Java
virtual machine port is complete. As almost all hardware and OS vendors have already licensed Java, it is expected that almost all-contemporary hardware will ultimately support both Java and JDCE.
Chapter 8 - Setting up JDCE for your desktop

Now that we have finished the discussion on JDCE widgets, a brief description of the steps involved in setting up the servers and running the clients is in order. We start this process by first describing the pre-requisites for running JDCE. We then detail the steps involved in starting the servers and setting up JDCE. Last, but not the least we describe the various applet tags for both the RMI and CORBA clients.

8.1 Pre-requisites

- A CORBA-compliant Java Object Request Broker.
- The CORBA Naming service.
- The JDK 1.1.X Virtual Machine.

8.2 Starting the servers & setting up JDCE

- Starting the Active Object Server, the default port it binds to is 14000 (ports less than 1024 require administrative privileges).

  % osagent

- Starting the Gatekeeper which acts as a HTTP-tunneler for Callbacks through firewalls, the default port it runs on is 15000

  % gatekeeper

- Starting up the RMIregistry on some port, (ports less than 1024 require
administrative privileges) say 7000. The RMI registry is a simple server-side bootstrap name server that allows remote clients to get a reference to a remote object. It is typically used only to locate the first remote object an application needs to talk to. That object in turn will provide application specific support for finding other objects.

% rmiregistry 7000

For security reasons, an application can bind or unbind only in the registry running on the same host. This prevents a client from removing or overwriting any of the entries in a server's remote registry. A lookup, however, can be done from any host. This requires every Client to start-up the rmiregistry on its host machine. In the case that the client and server are on the same machine, the rmiregistry needn't be started for the client. It is necessary to specify the port number only if a server creates a registry on a port other than the default 1099.

- Starting the Naming Service, with the nameContext Root as JDCE. To use the Name Service, at least one Naming Factory must be started. The Factory object lives within a server process, and is used to create NamingContext objects. When the default Factory server is started, it creates no NamingContexts. When it is asked to create NamingContexts, all such
NamingContext objects created by a given Factory are located within the same process as that Factory.

```java
%java -DORBservices=CosNaming -DSVCnameroot=JDCE -DJDKrenameBug com.visigenic.vbroker.services.CosNaming.ExtFactory JDCE namingLog
```

The first step to creating a namespace is starting at least one Factory server. Here we start up a single Factory server process. The Factory will be given the logical name "JDCE". In addition, the name of a log file is specified (in this case, the locally created "namingLog" file will be used). This log will contain information allowing the Name Service to be shut down and then reDOWNLOADd upon restart to its state prior to shutdown. Here we have used the idea of connecting Contexts that live within distinct Name Servers, the rootContext in this case is JDCE. Now that a Context has been created, we can "publish" an Object's reference within that Context. Anyone can access this published object if they have both:

- a reference to the NamingContext in which it is published.
- the name within that NamingContext with which the Object's reference was originally bound.

Starting the main Object Server, which starts up both the RMI/CORBA
sessionScheduler. The ORBservices System Property tells the ORB you wish your program to use the Name Service. The SVCnamerooot parameter tells the ORB which root naming context should be returned by resolve_initial_references. Now that a Context has been created, we can "publish" an Object's reference within that Context. Anyone can access this published object if they have both: a reference to the NamingContext in which it is published. the name within that NamingContext with which the Object's reference was originally bound.

%java -DORBservices=CosNaming -DSVCnamerooot=JDCE
jdce.impl.corba.sessionServer

### 8.3 Applet Tags for CORBA & RMI Clients.

Now that we have started up the servers, the only step that remains is executing the demonstration applets inside a browser or appletviewer. Besides the applet tag that is a pre-requisite for viewing any applet, we need differing tags for CORBA and RMI based applets. This section explains which, and why, additional tags are required. We start with the CORBA client and then proceed to the RMI client.

- The HTML file, for the JDCE CORBA-client applet. When using Communicator, you have two choices on how to program your applets. You may use the ORB embedded in Communicator (which matches VisiBroker for Java 2.5), or you may download the VisiBroker for Java 3.0 ORB on top of the existing version in Communicator. You should download the VisiBroker
for Java 3.0 ORB if you are using features new to VisiBroker 3.0 such as interceptors, event handlers, or SSL.

- To use the ORB runtime that comes with Communicator, you must run your servers and the GateKeeper in backward compatibility mode and add the following param tag to your applet tag in your HTML file.

  `<param name=USE_ORB_LOCATOR value=true>`

- To use VisiBroker for Java 3.0 instead of Communicator's version of VisiBroker, add the following param tag to your applet tag in your HTML file. No special flags are required for the GateKeeper or servers.

  `<param name=org.omg.CORBA.ORBClass value=com.visigenic.vbroker.orb.ORB>
  <param name=ORBservices value=CosNaming>
  <param name=SVCnameroot value=JDCE>
  </param>`

The following HTML file details the options discussed above for the CORBA based client.

```html
<applet code=demos.chat.ChatUser.class width=400 height=350>
  <param name=org.omg.CORBA.ORBClass value=com.visigenic.vbroker.orb.ORB>
  <param name=ORBservices value=CosNaming>
  <param name=SVCnameroot value=JDCE>
</applet>
```

- The HTML file, for the JDCE RMI-client applet. The `servername` and `serverPort` params specify the host and port number on which the server application is running. From the HTML file given below it is fair to say that the session server is hosted by `jojo.npac.syr.edu` and it is running on port `7000`.

```html
<applet code=demos.rmi.RMIChatUser.class width=400 height=350 serverHostname="jojo.npac.syr.edu" serverPort=7000>
</applet>
```
Chapter 9 - Developing JDCE Applications.

The goal of this section is to give a small demonstration of the techniques that are useful in building collaboration-aware applications grounds-up or to retrofit single user applications into multi-user applications. Of considerable significance is an understanding of the fact that Clients of remote objects interact with remote interfaces, never with the implementation classes of those interfaces. That aside, development of JDCE applications involves 3 basic steps in addition to the functionality provided by the application that you desire.

Getting a handle to the session Scheduler
Getting a reference to the DataBahn
Publishing the client Reference

Once these three steps have been completed successfully the client switches to collaboration mode with all other clients that have successfully registered with a certain DataBahn. Clients are never in collaboration mode with clients across dataBahn's, though there is a roundabout way of doing the same. To keep things simple we assume that clients are always in intra-Bahn-collaboration mode as opposed to the inter-Bahn-Collaboration mode.

9.1 Binding to the sessionScheduler (CORBA) Vs (RMI)

It is the sessionScheduler which returns a handle to existing dataBahns or provides for the creation of one. However, invocation of any of the aforementioned operations should
be preceded by the successful bind to the sessionScheduler, as is the case for invocations on any remote object. The RMI and CORBA implementations of the sessionScheduler provide the same application level semantics.

### 9.1.1 CORBA Binding Process.

In the CORBA binding process we need to first obtain the naming Contexts, create the composite name and then follow it up with location of the object based on its name. The Steps involved are detailed below:

- **Obtaining the Initial Naming Context:** The client obtains a reference to the initial naming context by invoking the `org.omg.CORBA.ORB`'s resolve_initial_references.

  ```java
  org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(this, null);
  org.omg.CORBA.Object nameServiceObj = orb.resolve_initial_references("NameService");
  ```

- **Creating the Composite name:** Since we are looking for Scheduler, we need to first create the compound name "Collaboration", "Scheduler".

  ```java
  NameComponent[] collabName = {
    new NameComponent("Collaboration", "Scheduler")
  };
  ```

- **Locate the Object with this Name:** Invoke `resolve` on the NamingContext to obtain the object reference associated with the name created in the earlier step. Since the Name Service returns generic CORBA objects, it is necessary to `narrow` the generic object to a more derived class.
private void getSessionSchedulerHandle() {
    try {
        /* Initialize the ORB. */
        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(this, null);

        /* Get a reference to the Naming Service */
        org.omg.CORBA.Object nameServiceObj =
            orb.resolve_initial_references("NameService");
        /* Check to see if the operation involving getting a reference to the NameService was successful */
        if (nameServiceObj == null) {
            System.out.println("Name Service Object = null");
            return;
        }

        org.omg.CosNaming.NamingContext nameService =
            org.omg.CosNaming.NamingContextHelper.narrow(nameServiceObj);

        if (nameService == null) {
            System.out.println("nameService = null");
            return;
        }

        NameComponent[] collabName =
            new NameComponent("Collaboration", "Scheduler");
        _chatSession =
            jdce.scheduler.sessionSchedulerHelper.narrow(nameService.resolve(collabName));
    } catch(Exception e) {
        System.out.println("Exception: " + e);
    }
}

9.1.2 RMI Binding Process

The Client gets a reference to the "Scheduler" from the server's registry by constructing the URL rmi://jojo.npac.syr.edu:7000/Scheduler, where jojo.npac.syr.edu represents the HOSTNAME of the machine hosting the server, and 7000 represents the port the server listens to. This URL is used as a parameter to the java.rmi.Naming.lookup() method, this returns a generic Remote Object which is then cast to jdce.scheduler.RMISessionScheduler.
9.2 Getting a handle to the Data Bahn (CORBA) Vs (RMI)

The dataBahn supports various messaging primitives as mentioned in prior sections. Besides this the dataBahn also maintains a list of all the clients registered to it and is responsible for creation of worker threads (when the client has successfully registered) to handle callbacks for individual clients. Reception of the dataBahn handle is a precursor to the registration process, which switches the client into collaboration mode.

9.2.1 The CORBA DataBahn Handler

The following code snippet details the process involved in the creation/joining of a DataBahn. As can be seen the process of creating a DataBahn throws a jdceBahnException. The steps involved in Binding to the DataBahn Object are identical to that of the session Scheduler Object, but for a minor change in the NameComponent, which in this case would be a "Collaboration", DataBahn_Name. It should be understood nevertheless that getting a handle to the DataBahn can be achieved in two ways, the first technique is detailed in the code below, the second one involves invoking the getDataBahnHandle() on the session Scheduler Object. I have indicated the latter approach for the RMI Case.

```java
public void getDataBahnHandle() {
    try {
        remoteObject = Naming.lookup("rmi://jojo.npac.syr.edu:7000/Scheduler");
    } catch (java.lang.Exception exec) {
        System.out.println("Unable to lookup created Scheduler");
    }
    _chatSession = (jdce.scheduler.RMISessionScheduler) remoteObject;
}
```
chatSession.createBahn(partyName, "Chat");
System.out.println("New Party created");
} catch (jdceBahnException e) {
    System.out.println("Exception in creations" + e);
} catch (Exception e) {
    e.printStackTrace();
}

long startTime = System.currentTimeMillis();
try {
    // Initialize the ORB
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(this, null);

    /* Get a reference to the Naming Service */
    org.omg.CORBA.Object nameServiceObj =
        orb.resolve_initial_references("NameService");

    if (nameServiceObj == null) {
        System.out.println("Name Service Object = null");
        return;
    }

    org.omg.CosNaming.NamingContext nameService =
        org.omg.CosNaming.NamingContextHelper.narrow(nameServiceObj);
    if (nameService == null) {
        System.out.println("nameService = null");
        return;
    }

    NameComponent[] collabName =
        { new NameComponent("Collaboration", partyName + "Chat" + "Coordinator")};

    _chatBahn =
        jdce.bytways.dataBahnHelper.narrow(nameService.resolve(collabName));
} catch (Exception e) {
    System.out.println("Exception: " + e);
}

long stopTime = System.currentTimeMillis();
System.out.println("Avg Ping to invoke = "
    + ((stopTime - startTime)/1000f) + " msecs");

9.2.2 The RMI DataBahn Handler

The following code snippet details the process involved in the creation/joining of a DataBahn. As can be seen the process of creating a DataBahn throws a jdceRMIBahnException in addition to the usual java.rmi.RemoteException which is thrown for any remote method invocation. The technique in getting a handle to the DataBahn, involves the invocation of the getDataBahnHandle() method. The
second approach which could have been used is the approach similar to the one used for getting a reference to the session Scheduler only that in this case the constructed URL would have been rmi://jojo.npac.syr.edu:7000/DataBahn_Name

```java
public void getDataBahnHandle() {
    try {
        _chatSession.createBahn(partyName, "Chat");
        System.out.println("New Party created");
    } catch (jdceRMIBahnException e) {
        System.out.println("Exception in creations" + e);
    } catch (Exception e) {
        e.printStackTrace();
    }

    // long startTime = System.currentTimeMillis();

    try {
        _chatBahn = _chatSession.getDataBahnHandle(partyName, "Chat");
    } catch (jdceRMIBahnException e) {
        System.out.println("Exception in creations" + e);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

9.3 Publishing the Client Reference (CORBA) Vs (RMI)
We are ready to publish a clients reference once we have successful receipt of the dataBahn’s handle (this could be a pre-existing one or a newly created one). The registration process provides the worker threads with the callback objects which is what makes collaboration possible. As soon as a client registers to the dataBahn all the members that have registered to the bahn receive a notification of the new client’s presence.

9.3.1 Publishing the Client Reference (CORBA)
The code snippet show below assumes that the client in this case is an applet. In that case the initialization of the ORB takes the applet sub-class as an argument. Next we publish the ClientControlImpl class (which implements the clientProxy
interface) to the ORB. A simplistic implementation of the ClientControlImpl class
would merely have a reference to the applet sub-class. This is followed up with
the invocation of the register() method on the dataBahn object, with this client
reference as an argument.

```java
try {
    /* Initialize the ORB. */
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(thisClient);
    /* Create the Client Implementation object. */
    ClientControlImpl control = new ClientControlImpl(thisClient);
    orb.connect(control);

    try {
        thisClient._chatBahn.register(thisClient.clientName,(jdce.client.clientProxy)control);
        System.out.println("Registration Succeeded");
    } catch (jdceBahnException e) {
        System.out.println(e.typeToString(e.getType()));
    } catch (jdceClientException e) {
        System.out.println("Registration Failed");
    }
}
```

9.3.2 Publishing the Client Reference (RMI)

The RMI publishing process is much simpler, and is immediately obvious
when one looks at the code snippet. In this case the reference which is
published implements the RMIClientProxy interface. A simplistic
implementation of the ClientControlImpl class would merely have a reference
to the applet sub-class. As mentioned before the dataBahns, in both the RMI
and CORBA case, provide for similar application level semantics, thus we use
the register() method to do the publishing.

```java
try {
    control = new RMIClientControlImpl(thisClient);
    identity = "rmi://hawaii.npac.syr.edu:7000/" + thisClient.clientName;
    Naming.rebind(identity, control);
    System.out.println("ClientControl Export done.");
} catch (RemoteException re) {
    System.out.println("Exception in Client Impl.main: " + re);
} catch (MalformedURLException e) {
    System.out.println("MalformedURLException CoordinatorImpl.main: " + e);
}
```
try {
    thisClient._chatBahn.register(thisClient.clientName, control);
    System.out.println("Registration Succeeded");
} catch (jdceRMIBahnException e) {
    System.out.println(e.typeToString(e.getType()));
    System.out.println("Registration Failed");
} catch (RemoteException e) {System.out.println("Remote Exception"+
    e.printStackTrace());}

9.4 The Development Of Collaboration aware Code

In this context I choose the "HelloWorld"-equivalent of Collaboratory applications - CHAT. I intend to explore both the RMI and CORBA versions of this application, and as a proof-of-concept detail the collaboration involving RMI & IIOP Chat Clients. This Chat Application does not utilize IRC for communication, but employs the JDCE remote object sharing mechanism to achieve the same effect. The participants can log on to different interest groups and communicate within the group

**CORBA-based Collaboration** The two files which are employed in building the demonstration Chat applets are

- ChatUser.java
- ClientControlImpl.java

These files have been listed in Appendix C

**RMI-based Collaboration** In this case the files involved are

- RMIChatUser.java
• **RMIClientControlImpl.java**

These files have been listed in Appendix D.

### 9.5 The Retrofit Process

Due to space constraints, and also the fact that the retrofit process for JDCE is almost similar to the collaboration-aware case, this section would be demonstrated in the actual presentation.
Chapter 10 – Summary of results for JDCE

_It is a capital mistake to theorize before one has data....... Insensibly one begins to twist facts to suit theories, instead of theories to suit facts._

SHERLOCK HOLMES
Sir Arthur Conan Doyle, *A Scandal in Bohemia* (1891)

The following chapter summaries the various results obtained for JDCE. We start, with timing comparisons for the JDCE initialization process, average response times during collaboration of only RMI/CORBA clients, and finally with a comparison of the response times when RMI & CORBA clients are in collaboration mode.

**10.1 The JDCE Initialization process**

As detailed in earlier chapters, the JDCE initialization process comprises of getting a reference to the session Scheduler, the creation/joining of a data Bahn, getting a reference to the data Bahn, and finally publishing the client reference and registering it with the data Bahn. The table below, indicates that the initialization time for the RMI clients is much higher than those for the CORBA clients, though it should be understood that in case of CORBA we are using the ORB classes provided with Netscape.

<table>
<thead>
<tr>
<th></th>
<th>RMI</th>
<th>CORBA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receipt of Session</td>
<td>3.5 s</td>
<td>1.587 s</td>
</tr>
<tr>
<td>Scheduler handle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Creation of Data Bahn</td>
<td>246 ms</td>
<td>122 ms</td>
</tr>
<tr>
<td>Receipt of Data Bahn</td>
<td>193 ms</td>
<td>115 ms</td>
</tr>
<tr>
<td>handle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Publishing Client reference</td>
<td>737 ms</td>
<td>498 ms</td>
</tr>
<tr>
<td>Register Client with the</td>
<td>599 ms</td>
<td>682 ms</td>
</tr>
<tr>
<td>Data Bahn</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
10.2 Response times for pure CORBA/RMI Collaboration modes

In this case we have only CORBA or only RMI clients in Collaboration mode. This helps us in giving a clearer picture of the response times before we embark on comparing the response times when we have both CORBA & RMI clients in collaboration mode. It should be understood nevertheless, that we are calculating the average time required for a client to broadcast and receive a set of messages. The implication here is the fact that if there are 2 groups and 2 clients, if there were 10 broadcast messages instantiated by a client in each group there would be 60 messages handled by the server. By a similar analogy if there were 8-clients on each group, and there are 500 messages sent by a client in each group there would be 10,000 messages handled by the server before the results are calculated.

MQA time in tables 4, 5, 6 refer to average time to add a single jdceMessage to the jdceMessage queue. The object roundtrip time refers to the time it takes to receive a message from the server, in response to a broadcast from one of the clients. All calculations have been performed 7 times each and an average has been calculated by ignoring the highest and lowest times, and calculating the average of the other 5 cases. This to some extent filters out any deviant behavior in the data.

Table 4 details the average response times for the case where we have 2 DataBahns with 2 clients, RMI/CORBA, in collaboration mode.
<table>
<thead>
<tr>
<th>Number of Messages sent</th>
<th>RMI</th>
<th>CORBA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MQA time per message</td>
<td>Object round trip per message</td>
</tr>
<tr>
<td>1</td>
<td>19 ms</td>
<td>36 ms</td>
</tr>
<tr>
<td>10</td>
<td>21 ms</td>
<td>67 ms</td>
</tr>
<tr>
<td>25</td>
<td>19 ms</td>
<td>47 ms</td>
</tr>
<tr>
<td>50</td>
<td>20 ms</td>
<td>52 ms</td>
</tr>
<tr>
<td>100</td>
<td>22 ms</td>
<td>70 ms</td>
</tr>
<tr>
<td>250</td>
<td>23 ms</td>
<td>79 ms</td>
</tr>
<tr>
<td>500</td>
<td>26 ms</td>
<td>99 ms</td>
</tr>
</tbody>
</table>

**Table 4:** JDCE response times for 2 DataBahns with 2 clients each

Table 5 details the average response times for the case where we have 2 DataBahns with 4 clients, RMI/CORBA, in collaboration mode.

<table>
<thead>
<tr>
<th>Number of Messages sent</th>
<th>RMI</th>
<th>CORBA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MQA time per message</td>
<td>Object round trip per message</td>
</tr>
<tr>
<td>1</td>
<td>21 ms</td>
<td>38 ms</td>
</tr>
<tr>
<td>10</td>
<td>20 ms</td>
<td>90 ms</td>
</tr>
<tr>
<td>25</td>
<td>21 ms</td>
<td>62 ms</td>
</tr>
<tr>
<td>50</td>
<td>21 ms</td>
<td>105 ms</td>
</tr>
<tr>
<td>100</td>
<td>21 ms</td>
<td>131 ms</td>
</tr>
<tr>
<td>250</td>
<td>24 ms</td>
<td>137 ms</td>
</tr>
<tr>
<td>500</td>
<td>28 ms</td>
<td>178 ms</td>
</tr>
</tbody>
</table>

**Table 5:** JDCE response times for 2 DataBahns with 4 clients each
Table 6 details the average response times for the case where we have 2 DataBahns with 2 clients, RMI/CORBA, in collaboration mode.

<table>
<thead>
<tr>
<th>Number of Messages sent</th>
<th>RMI</th>
<th>CORBA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MQA time per message</td>
<td>Object round trip per message</td>
</tr>
<tr>
<td>1</td>
<td>22 ms</td>
<td>60 ms</td>
</tr>
<tr>
<td>10</td>
<td>22 ms</td>
<td>89 ms</td>
</tr>
<tr>
<td>25</td>
<td>25 ms</td>
<td>89 ms</td>
</tr>
<tr>
<td>50</td>
<td>24 ms</td>
<td>148 ms</td>
</tr>
<tr>
<td>100</td>
<td>24 ms</td>
<td>180 ms</td>
</tr>
<tr>
<td>250</td>
<td>29 ms</td>
<td>213 ms</td>
</tr>
<tr>
<td>500</td>
<td>32 ms</td>
<td>234 ms</td>
</tr>
</tbody>
</table>

**Table 6** JDCE Response times with 2 DataBahns with 8 clients each

### 10.3 Response times for JDCE CORBA & RMI clients in collaboration mode

The figure below details the response times when CORBA & RMI clients are in collaboration mode. The legend requires a bit of explaining to do. RMI (2,2) and CORBA (2,2) corresponds to the case where there are 2 data Bahns, with one CORBA and one RMI client registered to each bahn. RMI(2, 8) and CORBA (2,8) corresponds to the case where there are two data Bahns with each Bahn comprising of 4 RMI and 4 CORBA clients.

The graph indicates the fact that when the CORBA and RMI clients are in collaboration
mode, with the respective worker threads being scheduled by the same thread scheduler, the RMI client receives the whole set of broadcast messages even though it was a CORBA client which initiated the broadcast. The graph also demonstrates that as the number of clients increases, as also the messages, the RMI based clients’ response times are almost 200% faster than that of the CORBA clients.
Figure 21: Response times - JDCE RMI & CORBA Clients in Collaboration mode
**Conclusions & Future Prognosis**

Hype, hyperbole, and failings aside, the CORBA software bus seems to offer a convenient framework for scalable multi-user collaboratory environments. CORBA offers both a potential candidate for universal wire protocol, IIOP, and a natural collaboratory framework based on shared CORBA objects and flexible message filtering mechanisms.

The JDCE RMI based Collaboration is about 80% faster than the IIOP solution, however the advantage, that RMI holds, is blunted by the fact that its a pure Java Solution. Java RMI is platform independent, albeit expressed through Java. Nevertheless, Java-CORBA is a platform independent and language independent solution. With CORBA one could have Java Helper classes accessing a C++ implementation of the DataBahn. The choice is clear in case of pure Java solutions write to RMI else write to CORBA.

However, scalability in such systems can be sustained only if some suitably defined communication locality is enforced. The most elaborate support for communication channel based filtering is to be provided soon by the HLA technology and the associated Run-Time Infrastructure (RTI) software bus model under development and standardization by DMSO. HLA builds on top of and extends DIS standards towards a truly interoperable simulation framework, including real-time and logical time simulations. HLA facilitates the division of simulation responsibilities between machines by manipulating attributes instead of entire objects. The HLA/RTI option seems to be the most promising future candidate for developing large scale massively scalable multi-user environments.
Appendix A: Interface Definition Language for the System

module java {
    module lang {
        extensible struct Object;
    };
};
module jdce {
    module util {
        module exception {
            exception jdceBahnException {
                long type;
            };
            exception jdceClientException {
                long type;
            };
        };
        module event {
            extensible struct jdceEvent {
                long type;
                string clientName;
            };
        };
        module message {
            extensible struct jdceMessage {
                string sender;
                string applicationType;
                string messageType;
                string protocolType;
                ::java::lang::Object content;
            };
        };
    };
};
module client {
    interface clientProxy {
        boolean clientEvent(
            in ::jdce::util::event::jdceEvent arg0
        );
        boolean dataReceive(
            in string arg0
        );
        boolean messageReceive(
            in ::jdce::util::message::jdceMessage arg0
        );
    };
};
module byteways {
    interface dataBahn {
        string getApplicationServerName();
        long numberOfMembers();
        typedef sequence sequence_of_string;
        ::jdce::byteways::dataBahn::sequence_of_string getClientNames();
        boolean isEmpty();
        long register(
            in string arg0,
            in ::jdce::client::clientProxy arg1
        )
        raises(
            ::jdce::util::exception::jdceBahnException,
            ::jdce::util::exception::jdceClientException
        );
boolean expelClient(
   in string arg0,
   in string arg1
)
raises(
   ::jdce::util::exception::jdceBahnException,
   ::jdce::util::exception::jdceClientException
);

boolean deregister(
   in string arg0
)
raises(
   ::jdce::util::exception::jdceClientException
);

boolean broadcast(
   in string arg0
);

boolean broadcastMessage(
   in ::jdce::util::message::jdceMessage arg0
)
raises(
   ::jdce::util::exception::jdceBahnException
);

boolean inform(
   in ::jdce::util::message::jdceMessage arg0
)
raises(
   ::jdce::util::exception::jdceBahnException
);

boolean whisper(
   in string arg0,
   in string arg1
)
raises(
   ::jdce::util::exception::jdceClientException,
   ::jdce::util::exception::jdceBahnException
);

boolean whisperMessage(
   in ::jdce::util::message::jdceMessage arg0,
   in string arg1
)
raises(
   ::jdce::util::exception::jdceClientException,
   ::jdce::util::exception::jdceBahnException
);

module scheduler {
   interface sessionScheduler {
      typedef sequence sequence_of_string;
      ::jdce::scheduler::sessionScheduler::sequence_of_string listApplications(
      )
raises(
   ::jdce::util::exception::jdceBahnException
);
      ::jdce::scheduler::sessionScheduler::sequence_of_string listBahns(
      in string arg0
)
raises(
   ::jdce::util::exception::jdceBahnException
);
      boolean createBahn(
      in string arg0,
      in string arg1
   )
   
};

};
raises(
    ::jdce::util::exception::jdceBahnException
);
boolean joinBahn(
    in string arg0,
    in string arg1
)
raises(
    ::jdce::util::exception::jdceBahnException
);
::jdce::byteways::dataBahn getDataBahnHandle(
    in string arg0,
    in string arg1
)
raises(
    ::jdce::util::exception::jdceBahnException
);
boolean destroyBahn(
    in string arg0,
    in string arg1
)
raises(
    ::jdce::util::exception::jdceBahnException
);
);
Appendix B: JDCE Class Hierarchy

- class java.lang.Object
  - class jdce.util.message.CircularLinkedList
  - class org.omg.CORBA.portable.ObjectImpl (implements org.omg.CORBA.Object)
    - class org.omg.CORBA.portable.Skeleton
      - class jdce.client._clientProxyImplBase (implements jdce.client._clientProxy)
        - class jdce.client._tie_clientProxy
      - class jdce.byteways._dataBahnImplBase (implements jdce.byteways.dataBahn)
        - class jdce.impl.corba.DataBahnImpl
        - class jdce.byteways._tie_dataBahn
      - class jdce.scheduler._sessionSchedulerImplBase (implements jdce.scheduler._sessionScheduler)
        - class jdce.impl.corba.SessionSchedulerImpl
        - class jdce.scheduler._tie_sessionScheduler
    - class jdce.client._portable_stub_clientProxy (implements jdce.client._clientProxy)
    - class jdce.byteways._portable_stub_dataBahn (implements jdce.byteways.dataBahn)
    - class jdce.client._st_clientProxy (implements jdce.client._clientProxy)
    - class jdce.byteways._st_dataBahn (implements jdce.byteways.dataBahn)
    - class jdce.scheduler._st_sessionScheduler (implements jdce.scheduler._sessionScheduler)
  - interface jdce.client.RMIClientProxy (extends java.rmi.Remote)
  - interface jdce.byteways.RMIDataBahn (extends java.rmi.Remote)
  - interface jdce.scheduler.RMISessionScheduler (extends java.rmi.Remote)
    - class java.rmi.server.RemoteServer
      - class java.rmi.server.UnicastRemoteObject
        - class jdce.impl.mi.DataBahnImpl (implements jdce.byteways.RMIDataBahn)
        - class jdce.impl.mi.SessionSchedulerImpl (implements jdce.scheduler.RMISessionScheduler)
  - class java.lang.Thread (implements java.lang.Runnable)
    - class jdce.util.message.jdceRMIWorker
    - class jdce.util.message.jdceWorker
  - class jdce.util.message.ThreadScheduler (implements java.lang.Runnable)
• class java.lang.Throwable (implements java.io.Serializable)
  • class java.lang.Exception
    • class org.omg.CORBA.UserException
      • class jdce.util.exception.jdceBahnException (implements java.io.Serializable, jdce.util.exception.jdceException)
      • class jdce.util.exception.jdceClientException (implements java.io.Serializable, jdce.util.exception.jdceException)
      • class jdce.util.exception.jdceRMIException (implements java.io.Serializable, jdce.util.exception.jdceException)
    • class jdce.util.exception.jdceRMIClientException (implements java.io.Serializable, jdce.util.exception.jdceException)
• interface jdce.client.clientProxy (extends org.omg.CORBA.Object)
• class jdce.client.clientProxyHelper
• class jdce.client.clientProxyHolder (implements org.omg.CORBA.portable.Streamable)
• interface jdce.client.clientProxyOperations
• interface jdce.byteways.dataBahn (extends org.omg.CORBA.Object)
• class jdce.byteways.dataBahnHelper
• class jdce.byteways.dataBahnHolder (implements org.omg.CORBA.portable.Streamable)
• interface jdce.byteways.dataBahnOperations
• class jdce.util.exception.jdceBahnExceptionHelper
• class jdce.util.exception.jdceBahnExceptionHolder (implements org.omg.CORBA.portable.Streamable)
• class jdce.util.exception.jdceClientExceptionHelper
• class jdce.util.exception.jdceClientExceptionHolder (implements org.omg.CORBA.portable.Streamable)
• class jdce.util.event.jdceEvent (implements java.io.Serializable)
• class jdce.util.event.jdceEventHelper
• class jdce.util.event.jdceEventHolder (implements org.omg.CORBA.portable.Streamable)
• interface jdce.util.exception.jdceException
• class jdce.util.message.jdceMessage (implements java.io.Serializable)
• class jdce.util.message.jdceMessageHelper
• class jdce.util.message.jdceMessageHolder (implements org.omg.CORBA.portable.Streamable)
• class jdce.util.message.jdceMessageQueue (implements jdce.util.message.jdceMessageQueueOperations)
• interface jdce.util.message.jdceMessageQueueOperations
- interface jdce.scheduler.sessionScheduler (extends org.omg.CORBA.Object)
- class jdce.scheduler.sessionSchedulerHelper
- class jdce.scheduler.sessionSchedulerHolder (implements org.omg.CORBA.portable.Streamable)
- interface jdce.scheduler.sessionSchedulerOperations
- class jdce.impl.corba.sessionServer
- class jdce.impl.rmi.sessionServer
Appendix C: The CORBA Based CHAT Client

```java
package demos.chat;

import java.awt.*;
import java.applet.Applet;
import java.net.*;
import java.io.*;
import org.omg.CosNaming.*;
import jdce.util.exception.*;
import org.omg.CORBA.UserException;

/**
* This is just a simple server/client based proof-of-concept
* implementation to make sure that the compensation classes have
* all the required API in them.
* Below is the Chat User program. The applet consists of three parts:
* - a means of setting a name for the user
* - a scrolling list of messages
* - a means of sending a new message
* @version Alpha 1.0 6th October 1997
* @author shrideep@npac.syr.edu
* @author Shrideep Pallickara
* @author NorthEast Parallel Architectures Center
*/

public class ChatUser extends Applet implements Runnable {
    Button setIDButton;
    Button setNameButton;
    Button sayButton;
    Button signOffButton;
    Button partyNameButton;
    Button test;
    Button connectButton;
    Button joinButton;
    Button whisperButton;
    Button quitButton;

    TextField idField;
    TextField nameField;
    TextField typeField;
    TextField partyField;
    TextField testField;
    TextArea messageArea;

    Choice clientList;
    Choice bahnList;
    public jdce.byteways.dataBahn _chatBahn;
    public jdce.scheduler.sessionScheduler _chatSession;

    /** The name of this Client. */
    public String clientName;
    public int clientID;

    /**The name of the Party*/
    public String partyName;
    public int partyID;
    public int stressTest=1;

    /** Indicates if the client is successfully connected to the server. */
    boolean connected = false;
}
/* Default setup, will mostly be overridden by attributes. */
int width = 250;
int height = 250;

private Thread running = null;
private Thread controlThread;
public String clientNumber="1";

public synchronized void
init() {
Panel p;
Font lbl;
GridBagLayout g;
GridBagConstraints c;

/* Get needed attributes from html page. */
if (getParameter("width") != null) {
    width = Integer.parseInt(getParameter("width"));
} else {
    width = 250;
}

if (getParameter("height") != null) {
    height = Integer.parseInt(getParameter("height"));
} else {
    height = 250;
}

lbl = new Font("Helvetica", Font.BOLD, 14);
g = new GridBagLayout();
c = new GridBagConstraints();
c.fill = GridBagConstraints.NONE;
c.weightx = 1.0;
c.gridwidth = GridBagConstraints.REMAINDER;
setLayout(g);
p = new Panel();
g.setConstraints(p, c);
add(p);
p.add(test = new Button("StressTest"));
test.disable();
p.add(testField = new TextField(4)).setFont(lbl);
testField.setText("1");
p.add(setNameButton = new Button("Client Name"));
setNameButton.disable();
p.add(nameField = new TextField(10)).setFont(lbl);

p = new Panel();
g.setConstraints(p, c);
add(p);
p.add(partyNameButton = new Button("Create Bahn"));
p.add(partyField = new TextField(20)).setFont(lbl);
p.add(quitButton = new Button("Quit"));

p = new Panel();
g.setConstraints(p, c);
add(p);
p.add(connectButton = new Button("Connect"));
p.add(joinButton = new Button("Join"));
p.add(bahnList =new Choice());
bahnList.addItem("Data Bahn List");
p = new Panel();
g.setConstraints(p, c);
add(p);
p.add(new Label("Messages: ") );
p.add(messageArea = new TextArea(6, 30 ));

p = new Panel();
g.setConstraints(p, c);
add(p);
p.add(sayButton = new Button("Say"));
sayButton.disable();
p.add(typeField = new TextField(20 ));
p.add(signOffButton = new Button("InformOthers"));

p = new Panel();
g.setConstraints(p, c);
add(p);
p.add(whisperButton = new Button("Whisper"));
whisperButton.disable();
p.add(clientList = new Choice());
clientList.addItem("Clients On Bahn");
signOffButton.disable();
resize(width, height);
show();
}

private boolean isConnected() {
System.out.println("ChatUser: isConnected: " + connected);
return(connected);
}

private boolean connect() {
controlThread = new ClientControlThread(this);
controlThread.start();
return true;
}

private void binder() {
try {
/* Initialize the ORB. */
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(this, null);

/* Get a reference to the Naming Service */
org.omg.CORBA.Object nameServiceObj =
orb.resolve_initial_references("NameService");
if (nameServiceObj == null) {
System.out.println("Name Service Object = null");
return;
}

org.omg.CosNaming.NamingContext nameService =
org.omg.CosNaming.NamingContextHelper.narrow(nameServiceObj);
if (nameService == null) {
System.out.println("nameService = null");
return;
}

NameComponent[] collabName =
{ new NameComponent("Collaboration", "Scheduler")};
_chatSession =
dice.scheduler.sessionSchedulerHelper.narrow(nameService.resolve(collabName));
} catch(Exception e) {
System.out.println("Exception: " + e);
}
public void createDataBahn() {
    try {
        _chatSession.createBahn(partyName, "Chat");
        System.out.println("New Party created");
    } catch (jdceBahnException e) {
        System.out.println("Exception in creations" + e);
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public void getDataBahnHandle() {
    try {
        /* Initialize the ORB */
        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(this, null);

        /* Get a reference to the Naming Service */
        org.omg.CORBA.Object nameServiceObj =
            orb.resolve_initial_references("NameService");
        if (nameServiceObj == null) {
            System.out.println("Name Service Object = null");
            return;
        }
        org.omg.CosNaming.NamingContext nameService =
            org.omg.CosNaming.NamingContextHelper.narrow(nameServiceObj);
        if (nameService == null) {
            System.out.println("nameService = null");
            return;
        }
        NameComponent[] collabName =
            { new NameComponent("Collaboration", partyName+"Chat"+"Coordinator")};
        _chatBahn =
            jdce.byeways.dataBahnHelper.narrow(nameService.resolve(collabName));
    } catch(Exception e) {
        System.out.println("Exception: " + e);
    }
}

/* Convert the users message to bytes and send it over the DataBahn. */
public boolean broadcast(String message) {
    jdce.util.message.jdceMessage _message =
        new jdce.util.message.jdceMessage();
    _message.setContent((Object)getField.getText());
    _message.setSender(clientName);
    _message.setBahnName(partyName+"Chat"+"Bahn");
    try {
        for (int i=0; i < stressTest; i++) {
            _message.setContent((Object)getField.getText());
            _chatBahn.broadcastMessage(_message);
        }
    } catch(jdceBahnException e) {
        System.out.println("Exception in CLIENT Broadcast" + e);
    }
    return true;
}

public boolean dataReceive(String Message) {
    int position;
String message = Message + "\n";
position = messageArea.getText().length();
messageArea.insertText(message, position);
return true;
}

public boolean messageReceive(jdce.util.message.jdceMessage Message) {
    int position;
    String message;
synchronized(Message) {
        message = (String)Message.getContent() + "\n";
    }
    position = messageArea.getText().length();
    messageArea.insertText(message, position);
    return true;
}

public void informOthers() {
    jdce.util.message.jdceMessage _mess = new jdce.util.message.jdceMessage();
    _mess.setSender(clientName);
    _mess.setBahnName(partyName + "Chat" + "Bahn");
    try {
        for (int i = 0; i < stressTest; i++) {
            _mess.setContent((Object)typeField.getText() + " Inform" + i);
            _chatBahn.inform(_mess);
        }
    } catch (jdceBahnException e) {}
}

public void whisperToClient() {
    if (clientList.getSelectedIndex() == 0) {
        messageArea.setText("Please select a valid Client");
        return true;
    }
    try {
        _chatBahn.whisper(clientName + " Whisper:" + typeField.getText(),
                           clientList.getSelectedItem());
        return true;
    } catch (jdceBahnException e) {}
    catch (jdceClientException e) {}
}

private void disconnect() {
    System.out.println("Deregistring Client " + clientName);
    try {
        _chatBahn.deregister(clientName);
    } catch (jdceClientException e) {
        System.out.println("Error in deregistring");
    }
}

public void listMembers() {
    int numberOfMembers = _chatBahn.numberOfMembers();
    String[] temp = new String[numberOfMembers];
    if (numberOfMembers != 0) {
        System.arraycopy(_chatBahn.getClientNames(), 0, temp, 0, numberOfMembers);
        clientList.removeAll();
        clientList.addItem("Clients On Bahn");
        for (int i = 0; i < numberOfMembers; i++) {
            System.out.println(temp[i]);
            clientList.addItem(temp[i]);
        }
    }
    System.out.println("Listing complete");
}
public void listDataBahns() {
    int numberOfMembers;
    String[] temp;
    try {
        numberOfMembers = _chatSession.listBahns("Chat").length;
        temp = new String[numberOfMembers];
        System.arraycopy(_chatSession.listBahns("Chat").0, temp, 0, numberOfMembers);
        bahnList.removeAll();
        bahnList.addItem("DataBahn List");
        for (int i = 0; i < numberOfMembers; i++) {
            System.out.println(temp[i]);
            bahnList.addItem(temp[i]);
        }
    } catch (jdce.util.exception.jdceBahnException e) {
        System.out.println("Error in listing");
    }
}

public void start() {
    System.out.println("ChatUser: start.");
    if (running == null) {
        running = new Thread(this);
        running.start();
    }
}

public void stop() {
    if (running != null) {
        running.stop();
        running = null;
    }
}

public void run() {
}

public void destroy() {
    System.out.println("destroyed session");
}

public boolean action(Event event, Object obj) {
    System.out.println("ChatUser: action.");
    if ("InformOthers".equals(obj)) {
        informOthers();
        return true;
    }
    if ("Say".equals(obj)) {
        return broadcast(clientName + ":" + typeField.getText());
    }
    if ("StressTest".equals(obj)) {
        try {
            stressTest = Integer.parseInt(testField.getText());
        } catch (java.lang.NumberFormatException e) {
        }
    }
}
stressTest=1;
"
return true;
}
if ("SetID".equals(obj)) {
    clientNumber = idField.getText();
}
if ("Connect".equals(obj)) {
    binder();
    listDataBahns();
    return true;
}
if("Join".equals(obj)) {
    if (bahnList.getSelectedIndex()==0) {
        messageArea.setText("Please select a valid Bahn Name");
        return true;
    }
    if (bahnList.getSelectedIndex()==0) {
        messageArea.setText("Please select a valid Bahn Name");
        return true;
    }
    partyName = bahnList.getSelectedItem();
    partyName = partyName.substring(0, partyName.length() -4);
    System.out.println(partyName);
    getDataBahnHandle();
    listMembers();
    setNameButton.enable();
    return true;
}
if ("Create Bahn".equals(obj)) {
    partyName=partyField.getText();
    createDataBahn();
    getDataBahnHandle();
    System.out.println("App Server Name" + _chatBahn.getApplicationServerName());
    setNameButton.enable();
    return true;
}
if("Whisper".equals(obj)) {
    whisperToClient();
    return true;
}
if("Quit".equals(obj)) {
    disconnect();
    System.exit(0);
}
if ("Client Name".equals(obj)) {
    clientName = nameField.getText();
    if (clientNumber == null || clientName==null || partyName==null) {
        nameField.setText("Need to give a name!"("Client Name"));
        return(true);
    } else {
        connected = connect();
        signOffButton.enable();
        sayButton.enable();
        test.enable();
        whisperButton.enable();
        return(true);
    }
    return(false);
    }
}/*This thread is responsible for the callbacks from the Server side */
class ClientControlThread extends Thread {
    private ChatUser thisClient;
    private jdce.byteways.dataBahn_chatBahn;
    private jdce.scheduler.sessionScheduler_chatSession;

    ClientControlThread(ChatUser client) {
        thisClient = client;
    }

    public void run() {
        try {
            /* Initialize the ORB. */
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(thisClient);
            /* Create the ClientControl object. */
            ClientControlImpl control = new ClientControlImpl(thisClient);
            orb.connect(control);

            try {
                /* Get a reference to the Naming Service */
                org.omg.CORBA.Object nameServiceObj =
                    orb.resolve_initial_references("NameService");
                if (nameServiceObj == null) {
                    System.out.println("Name Service Object = null");
                    return;
                }

                org.omg.CosNaming.NamingContext nameService =
                    org.omg.CosNaming.NamingContextHelper.narrow(nameServiceObj);
                if (nameService == null) {
                    System.out.println("nameService = null");
                    return;
                }

                NameComponent[] collabName =
                    new NameComponent("Collaboration",
                    thisClient.partyName + "Chat" + "Coordinator");
                _chatBahn = jdce.byteways.dataBahnHelper.narrow(nameService.resolve(collabName));
            } catch(Exception e) {
                System.out.println("Exception: "+ e);
            }

            try {
                thisClient._chatBahn.register(thisClient.clientName,
                    (jdce.client.clientProxy)control);
            } catch(jdceBahnException e) {
                System.out.println(e.typeToString(e.getType()));
                System.out.println("Registration Failed");
            } catch(jdceClientException e) {}

            System.out.println("Control: Waiting for requests");
            try {
                Thread.currentThread().join();
            } catch(InterruptedException e) {
                System.out.println("error" + e);
            }
        } catch(org.omg.CORBA.SystemException e) {
            System.err.println(e);
        }
    }
}

package demos.chat;
public class ClientControlImpl extends jdce.client._clientProxyImplBase{
    private ChatUser thisClient;

    ClientControlImpl(Object client)
    {
        super();
        thisClient = (ChatUser)client;
        System.out.println("Client Control Object Created as client" + thisClient.clientNumber);
    }

    public boolean dataReceive(String Message) {
        return thisClient.dataReceive(Message);
    }

    public boolean messageReceive(jdce.util.message.jdceMessage Message) {
        return thisClient.messageReceive(Message);
    }

    public boolean clientEvent(jdce.util.event.jdceEvent evt) {
        System.out.println(evt.typeToString());
        return true;
    }
}
Appendix D: The RMI Based CHAT Client

package demos.rmi;
import java.awt.*;
import java.applet.Applet;
import java.net.*;
import java.io.*;
import jdce.util.exception.*;
import java.rmi.*;
import java.rmi.server.*;
import java.awt.image.*;
import java.awt.*;

/**
 * This is just a simple server/client based proof-of-concept
 * implementation to make sure that the compensation classes have
 * all the required API in them. </i></b>
 *
 * Below is the Chat User program. The applet consists of three parts:
 * - a means of setting a name for the user
 * - a scrolling list of messages
 * - a means of sending a new message
 *
 * @version     Alpha 1.0 6th October 1997
 * @author      shrideep@npac.syr.edu
 * @author      Shrideep Pallickara
 * @author      NorthEast Parallel Architectures Center
 */

public class RMIChatUser extends Applet implements Runnable, Remote {
    Button setIDButton;
    Button setNameButton;
    Button sayButton;
    Button signOffButton;
    Button partyNameButton;
    Button test;
    Button clientHostButton;
    Button clientPortButton;
    Button connectButton;
    Button joinButton;
    Button whisperButton;
    Button quitButton;
    TextField clientHostField;
    TextField clientPortField;
    TextField idField;
    TextField nameField;
    TextField typeField;
    TextField partyField;
    TextField testField;
    TextArea messageArea;
    Choice clientList;
    Choice bahnList;

    public jdce.byeways.RMIDataBahn _chatBahn;
    public jdce.scheduler.RMISessionScheduler _chatSession;
    private Remote remoteObject;

    /** The name of this Client. */
    public String clientName;
    public int clientID;
    /**The name of the Party*/
}
public String partyName;
public int partyID;

/** Indicates if the client is successfully connected to the server. */
boolean connected = false;

/** Default setup, will mostly be overridden by attributes. */
int width = 250;
int height = 250;
String hostname = "dodo.npac.syr.edu";
int port = 9999;
String clientHostname;
int clientPort;
public int stressTest=1;

private Thread running = null;
private Thread controlThread;
public String clientNumbers="1";

String testerString="Check for transfer file mode";
static int messageNumber=0;

private long startTime, stopTime;

public synchronized void init() {
    Panel p;
    Font lbl;
    GridBagLayout g;
    GridBagConstraints c;

    /* Get needed attributes from the html page. */
    if (getParameter("width") != null) {
        width = Integer.parseInt(getParameter("width"));
    } else {
        width = 250;
    }

    if (getParameter("height") != null) {
        height = Integer.parseInt(getParameter("height"));
    } else {
        height = 250;
    }

    if (getParameter("serverHostname") != null) {
        hostname = getParameter("serverHostname");
    } else {
        hostname = "dodo.npac.syr.edu";
    }

    if (getParameter("serverPort") != null) {
        port = Integer.parseInt(getParameter("serverPort"));
    } else {
        port = 250;
    }

    lbl = new Font("Helvetica", Font.BOLD, 14);
    g = new GridBagLayout();
    c = new GridBagConstraints();
    c.fill = GridBagConstraints.NONE;
    c.weightx = 1.0;
    c.gridwidth = GridBagConstraints.REMAINDER;
    g.setLayout(g);
    p = new Panel();
g.setConstraints(p, c);
add(p);
p.add(clientHostButton = new Button("ClientHost"));
p.add(clientHostField = new TextField(30)).setFont(lbl);

p = new Panel();
g.setConstraints(p, c);
add(p);
p.add(clientPortButton = new Button("ClientPort"));
p.add(clientPortField = new TextField(5)).setFont(lbl);
p.add(setNameButton = new Button("Client Name"));
setNameButton.disable();
p.add(nameField = new TextField(10)).setFont(lbl);

p = new Panel();
g.setConstraints(p, c);
add(p);
p.add(partyNameButton = new Button("Create Bahn"));
partyNameButton.disable();
p.add(partyField = new TextField(10)).setFont(lbl);
p.add(signOffButton = new Button("InformOthers"));
signOffButton.disable();

p = new Panel();
g.setConstraints(p, c);
add(p);
p.add(quitButton = new Button("Quit"));
quitButton.disable();
p.add(connectButton = new Button("Connect"));
p.add(joinButton = new Button("Join"));
joinButton.disable();
p.add(bahnList =new Choice());
bahnList.addItem("Data Bahn List");

p = new Panel();
g.setConstraints(p, c);
add(p);
p.add(new Label("Messages: "));
p.add(messageArea = new TextArea(6, 30));

p = new Panel();
g.setConstraints(p, c);
add(p);
p.add(sayButton = new Button("Say"));
sayButton.disable();
p.add(typeField = new TextField(20));
p.add(test = new Button("StressTest"));
test.disable();
p.add(testField = new TextField(4)).setFont(lbl);

p = new Panel();
g.setConstraints(p, c);
add(p);
p.add(whisperButton = new Button("Whisper"));
whisperButton.disable();
p.add(clientList = new Choice());
clientList.addItem("Clients On Bahn");
signOffButton.disable();
resize(width, height);
show();
}

private boolean isConnected() {
System.out.println("RMIChatUser: isConnected: " + connected);
return(connected);
}
private boolean connect() {
    controlThread = new ClientControlThread(this);
    controlThread.start();
    return true;
}

private void binder() {
    try {
        remoteObject = Naming.lookup("rmi://" + hostname + ":" + port + "/Scheduler");
    } catch (java.lang.Exception exec) {
        System.out.println("Unable to lookup created Scheduler");
    }
    _chatSession = (jdce.scheduler.RMISessionScheduler) remoteObject;
}

public void createDataBahn() {
    try {
        _chatSession.createBahn(partyName, "Chat");
        System.out.println("New Party created");
    } catch (jdceRMIBahnException e) {
        System.out.println("Exception in creations" + e);
        e.printStackTrace();
    }
}

public void getCoordinatorHandle() {
    try {
        _chatBahn = _chatSession.getDataBahnHandle(partyName, "Chat");
    } catch (jdceRMIBahnException e) {
        System.out.println("Exception in creations" + e);
        e.printStackTrace();
    }
}

/* Convert the users message to bytes and send it over the channel. */

public boolean broadcast(String message) {
    jdce.util.message.jdceMessage _message =
        new jdce.util.message.jdceMessage();
    _message.setContent((Object)typeField.getText());
    _message.setSender(clientName);
    _message.setBahnName(partyName + "Chat" + "Bahn");
    try {
        for (int i=0; i < stressTest; i++) {
            _message.setContent((Object)(typeField.getText()));
            _chatBahn.broadcastMessage(_message);
        }
    } catch (jdceRMIBahnException e) {
        System.out.println("Exception in CLIENT Broadcast" + e);
    } catch (RemoteException e) {
        System.out.println("Remote Exception" + e);
    }
    return true;
}

public boolean dataReceive(String Message) {
int     position;
String message=  Message + "\n";
position = messageArea.getText().length();
messageArea.insertText(message, position);
return true;
}

public boolean messageReceive(jdce.util.message.jdceMessage Message) {
    int     position;
    String message;
synchronized(Message) {
        message=  (String)Message.getContent() + "\n";
    }
    position = messageArea.getText().length();
    messageArea.insertText(message, position);
    return true;
}

public void informOthers() {
    jdce.util.message.jdceMessage _mess =
    new jdce.util.message.jdceMessage();
    _mess.setSender(clientName);
    _mess.setBahnName(partyName+"Chat*"+"Bahn");
    try {
        for (int i=0; i < stressTest; i++) {
            _mess.setContentView(typeField.getText() + " Inform" + i);
            _chatBahn.inform(_mess);
        }
    }
    catch (jdceRMIBahnException e) {}
    catch (RemoteException e) {
        System.out.println("Remote Exception " + e);
    }
}

public void whisperToClient() {
    if (clientList.getSelectedIndex()==0) {
        messageArea.setText("Please select a valid Client");
        return true;
    }
    try {
        for (int i=0; i < stressTest; i++) {
            _chatBahn.whisper(clientName + " Whisper:" +typeField.getText(),
                               clientList.getSelectedItem());
        }
    }
    catch (jdceRMIBahnException e) {
        System.out.println(e.typeToString(e.getType()));
    }
    catch (jdceRMIClientException e) {
        System.out.println(e.typeToString(e.getType()));
    }
    catch (RemoteException e) {
}}

private void disconnect() {
    try {
        _chatBahn.deregister(clientName);
        System.out.println("Deregistring Client " + clientName);
    }
    catch (jdceRMIClientException e) {
        System.out.println("Error in deregistring");
    }
    catch (RemoteException e) {
}}
public void listMembers() {
    int numberOfMembers=10;
    try {
        numberOfMembers=_chatBahn.numberOfMembers();
    } catch (RemoteException e) {
        System.out.println("Remote Exception " + e);
    }
    String[] temp = new String[numberOfMembers];
    if (numberOfMembers !=0) {
        try {
            System.arraycopy(_chatBahn.getClientNames(),0, temp,0,numberOfMembers);
        } catch (RemoteException e) {
            System.out.println("Remote Exception " + e);
        }
        clientList.removeAll();
        clientList.addItem("Clients On Bahn");
        for (int i=0;i< numberOfMembers; i++) {
            System.out.println(temp[i]);
            clientList.addItem(temp[i]);
        }
        System.out.println("Listing complete");
    }
}

public void listDataBahns() {
    int numberOfMembers;
    String[] temp;
    try {
        numberOfMembers = _chatSession.listBahns("Chat").length;
        temp = new String[numberOfMembers];
        System.arraycopy(_chatSession.listBahns("Chat"), 0, temp,0, numberOfMembers);
        bahnList.removeAll();
        bahnList.addItem("DataBahn List");
        for (int i=0;i< numberOfMembers; i++) {
            System.out.println(temp[i]);
            bahnList.addItem(temp[i]);
        }
    } catch (jdce.util.exception.jdceRMIBahnException e) {
        System.out.println("Error in listing");
    } catch (RemoteException e) {
        System.out.println("Remote Exception " + e);
    }
}

public void start() {
    System.out.println("RMIChatUser: start.");
    if (running == null) {
        running = new Thread(this);
        running.start();
    }
}

public void stop() {
    if (running != null) {
        running.stop();
        running = null;
    }
}
public void run() {
}

public void destroy() {
    System.out.println("destroyed session");
}

public boolean action(Event event, Object obj) {
    System.out.println("RMIChatUser: action.");

    if ("InformOthers".equals(obj)) {
        informOthers();
        return true;
    } else if ("Say".equals(obj)) {
        return broadcast(clientName + ":" + typeField.getText());
    } else if ("StressTest".equals(obj)) {
        try {
            stressTest = Integer.parseInt(testField.getText());
        } catch (java.lang.NumberFormatException e) {
            stressTest = 1;
        }
    } else if ("Whisper".equals(obj)) {
        whisper();
        return true;
    } else if ("Quit".equals(obj)) {
        disconnect();
        System.exit(0);
    } else if ("Connect".equals(obj)) {
        binder();
        listDataBahns();
        partyNameButton.enable();
        joinButton.enable();
        return true;
    } else if ("Join".equals(obj)) {
        if (bahnList.getSelectedIndex() == 0) {
            messageArea.setText("Please select a valid Bahn Name");
            return true;
        }
        partyName = bahnList.getSelectedItem();
        partyName = partyName.substring(0, partyName.length() - 4);
        System.out.println(partyName);
        getDataBahnHandle();
        listMembers();
        setNameButton.enable();
        return true;
    } else if ("Create Bahn".equals(obj)) {
        partyName = partyField.getText();
        createDataBahn();
        getDataBahnHandle();
        setNameButton.enable();
        listMembers();
        try {
            System.out.println("App Server Name" + _chatBahn.getApplicationServerName());
        } catch (RemoteException e) {
            System.out.println("Remote Exception " + e);
        }
    }
System.out.println("Received Party Handle");
return true;
}

if ("ClientHost".equals(obj)) {
if (clientHostField.getText()==null) {
    clientHostname=clientHostField.getText();
} else {
    return(true);
}

if ("Client Port".equals(obj)) {
if (clientPortField.getText()==null) {
    clientPort=integer.parseInt(clientPortField.getText());
} else {
    return(true);
}

if ("Client Name".equals(obj)) {
    if (clientNumber == null || clientName==null || partyName==null) {
        nameField.setText(" Need to give a name + ID before " + "Set Name");
        return(true);
    } else {
        connected = connect();
signOffButton.enable();
sayButton.enable();
test.enable();
whisperButton.enable();
quitButton.enable();
        return(true);
    }
return(false);
}
}

/*This thread is responsible for the callbacks from the Server side */

class ClientControlThread extends Thread {
private RMIChatUser thisClient;
private jdce.byways.RMIDataBahn _chatBahn;
private jdce.scheduler.RMISessionScheduler _chatSession;
private jdce.client.RMIClientProxy control;
String identity;
ClientControlThread(RMIChatUser client) {
    thisClient = client;
}

public void run() {
try {
    control = new RMIClientControllImpl(thisClient);
    identity = "rmi://" + thisClient.clientHostname +":"+thisClient.clientPort +"/"+thisClient.clientName;
    Naming.rebind(identity, control);
    System.out.println("ClientControl Export done.");
} catch (RemoteException re) {
    System.out.println("No luck dude" + re);
} catch (MalformedURLException e) {
    System.out.println("MalformedURLException CoordinatorImpl.main: " + e);
}
}
try {
    thisClient._chatBahn.register(thisClient.clientName, control);
} catch (jdceRMIChatException e) {
    System.out.println(e.typeToString(e.getType()));
    System.out.println("Registration Failed");
} catch (jdceRMIChatException e){}
catch (RemoteException e) {System.out.println("OPOOOOuch!!");}
System.out.println("Control: Waiting for requests");
}

package demos.rmi;

import java.rmi.*;
import java.rmi.server.*;
import java.net.*;

public class RMIClientControlImpl extends UnicastRemoteObject implements jdce.client.RMIClientProxy {
    private RMIChatUser thisClient;
    RMIClientControlImpl(Object client) throws java.rmi.RemoteException {
        super();
        thisClient = (RMIChatUser)client;
        System.out.println("Client Control Object Created as client" + thisClient.clientNumber);
    }

    public boolean dataReceive(String Message) throws java.rmi.RemoteException{
        return thisClient.dataReceive(Message);
    }

    public boolean messageReceive(jdce.util.message.jdceMessage Message) throws java.rmi.RemoteException {
        return thisClient.messageReceive(Message);
    }

    public boolean clientEvent(jdce.util.event.jdceEvent evt) throws java.rmi.RemoteException{
        System.out.println(evt.typeToString());
        return true;
    }
}
Glossary Of Terms

Applets
An application interface where referencing (perhaps by a mouse click) a remote application as a hyperlink to a server causes it to be downloaded and run on the client.

attribute
That part of an IDL interface that is similar to a public class field or C++ data member. The idltojava compiler maps an OMG IDL attribute to accessor and modifier methods in the Java programming language. CORBA attributes correspond closely to JavaBeans properties.

avatar
Geometry within a VRML scene that represents the user. The location of the avatar corresponds to the users viewing position

behaviours
A general term in VRML 2.0 to animate objects and to specify interactions among objects through output events, input events. routes, sensors and scripts.

CGI
Common-Gateway-Interface
A method for communication between a browser and a server for processing user input through a script and generating output that is sent back to the user. This script is usually placed in the cgi-bin directory of the server.

client
Any code which invokes an operation on a distributed object. A client might itself be a CORBA object, or it might be a non-object-oriented program, but while invoking a method on a CORBA object, it is said to be acting as client.

client stub
A Java programming language class generated by idltojava and used transparently by the client ORB during object invocation. The remote object reference held by the client points to the client stub. This stub is specific to the IDL interface from which it was generated, and it contains the information needed for the client to invoke a method on the CORBA object that was defined in the IDL interface.

COM (Common Object Model)
Microsoft's windows object model, which is being extended to distributed systems and multi-tiered architectures. ActiveX controls are an important class of COM object, which implement the component model of software. The distributed version of COM used to be called DCOM.

Component
An object which can be modified and reused without prior knowledge of the Implementation details.

ComponentWare
An approach to software engineering with software modules developed as objects with particular design frameworks (rules for naming and module architecture) and with visual editors both to interface to properties of each module and also to link modules together.

Component Model
The latest tidal wave in Object Oriented Programming which facilitates the reuse of objects without exposing any implementation details whatsoever to the end-user or programmer.

CORBA (Common Object Request Broker Architecture)
An approach to cross-platform cross-language distributed object developed by a broad industry group, the OMG. CORBA specifies basic services (such as naming, trading, persistence) the protocol IIOP used by communicating ORBS, and is developing higher level facilities which are object architectures for specialized domains such as
banking.

**CORBA Facility**
CORBA Facilities, occupy the conceptual void between the enabling technology defined by CORBA and Object Services, and the application-specific services, that the OMA labels "Application Objects".

**CORBA Horizontal Facility**
These Facilities are shared by many or most systems. There are four major sets of these facilities: User Interface, Information Management, Systems Management and Task Management.

**CORBA Vertical Facility**
Supports domain-specific tasks that are associated with vertical market segments.

**CORBA Services**
While the ORB specifies a systems grammar, Object Services represent its most basic vocabulary; the essential interfaces needed to create an object, introduce it into its environment, use and modify its features, and so forth. These services, bundled with every ORB, constitute the basic enabling technology of an OMA-compliant software system.

**Data Mining**
This describes the search and extraction of unexpected information from large databases. In a database of credit card transactions, conventional database search will generate monthly statements for each customer. Data mining will discover using ingenious algorithms, a linked set of records corresponding to fraudulent activity.

**Distributed Computing**
The use of networked heterogeneous computers to solve a single problem. The nodes (individual computers) are typically tightly coupled.

**Distributed Computing Environments**
The OSF Distributed Computing Environment (DCE) is a comprehensive, integrated set of services that supports the development, use and maintenance of distributed applications. It provides a uniform set of services, anywhere in the network, enabling applications to utilize the power of a heterogeneous network of computers, [http://www.osf.org/dce](http://www.osf.org/dce)

**Enterprise JavaBeans**
A cross-platform component architecture for the development and deployment of multi-tier, distributed, scalable, object-oriented Java applications.

**HLA (High Level Architecture)**
The High Level Architecture provides an architecture for modeling and simulation. The intent is to foster the interoperability of simulations (federation) and the re-use of simulation components.

**HPcc (High Performance commodity computing)**
NPAC project to develop a commodity computing based high performance computing software environment. Note that we have dropped "communications" referred to in the classic HPCC acronym. This is not because it is unimportant but rather because a commodity approach to high performance networking is already being adopted. We focus on high level services such as programming, data access and visualization that we abstract to the rather wishy-washy "computing" in the HPcc acronym.

**HPCC (High Performance Computing and Communication)**
Originally a formal federal initiative but even after this ended in 1996, this term is used to describe the field devoted to solving large-scale problems with powerful computers and networks.

**HPDC (High Performance Distributed Computing)**
The use of distributed networked computers to achieve high performance on a single problem, that is the computers are coordinated and synchronized to achieve a common goal.

**HPF (High Performance Fortran)**
A language specification published in 1993 by experts in compiler writing and parallel computation, the aim of which is to define a set of directives which allow a Fortran 90 program to run efficiently on distributed memory machines.

**HPJava**
An initiative by ARPA Parallel Compiler Runtime Consortium to address the use of Java for high performance language compilers and interpreters.

**Hyperlink**
The user-level mechanism (remote address specified in a HTML or VRML object) by which remote services are accessed by *Web Clients or Web Servers*.

**HTML (Hypertext Markup Language)**
A syntax for describing documents to be displayed on the *World Wide Web*.

**HTTP (Hyper Text Transport Mechanism)**
A stateless transport protocol allowing control information and data to be transmitted between web clients and servers. HTTP is the de facto standard for Web Communication.

**idltojava compiler**
A tool that takes an interface written in OMG IDL and produces Java programming language interfaces and classes that represent the mapping from the IDL interface to the Java programming language. The resulting files are.java files.

**IIOP (Internet Inter Orb Protocol)**
A stateful protocol allowing CORBA ORB's to communicate with each other, and transfer both the request for a desired service and the returned result.

**implementation**
A concrete class that defines the behavior for all of the operations and attributes of the IDL interface it supports. A servant object is an instance of an implementation. There may be many implementations of a single interface.

**initial naming context**
The NamingContext object returned by a call to the method orb.resolve_initial_references("NameService"). It is an object reference to the COS Naming Service registered with the ORB and can be used to create other NamingContext objects. See also: naming context

**Internet**
A complex set of interlinked national and global networks using the IP messaging protocol, and transferring data, electronic mail, and World Wide Web.

**invocation**
The process of performing a method call on a CORBA object, which can be done without knowledge of the object's location on the network. Static invocation, which uses a client stub for the invocation and a server skeleton for the service being invoked, is used when the interface of the object is known at compile time. If the interface is not known at compile time, dynamic invocation must be used.

**IP (Internet Protocol)**
The network-layer communication protocol used in the DARPA Internet. IP is responsible for host-to-host addressing and routing, packet forwarding, and packet fragmentation and reassembly.

**I-WAY**
An experimental high-performance ATM network linking dozens of the country's fastest computers and advanced visualization environments.

**Java**
An object-oriented interpreted programming language from Sun, suitable for Web development due to the built-in portable support for mobility, networking, multithreading and GUI.
Java API and Frameworks
A collection of packages including Java foundation classes and offering Java language interface to Java virtual machine (JavaVM). A Java Framework such as JDBC defines the standard Java methods and interfaces for accessing a particular service such as databases, commerce, graphics, multimedia etc. Such frameworks are determined by consensus in each area and are mirrored in CORBA as Facilities.

Java Applet API
an applet package in the Java API, the first in the long series of new APIs.

JavaBeans
Part of the Java 1.1 enhancements defining design frameworks (particular naming conventions) and inter JavaBean communication mechanisms for Java components with standard (Bean box) or customized visual interfaces (property editors). Enterprise JavaBeans are JavaBeans enhanced for server side operation with capabilities such as multi user support. JavaBeans are Java's component technology and in this sense are more analogous to ActiveX than either COM or CORBA. However JavaBeans augmented with RMI can be used to build a "pure Java" distributed object model.

Java Blend
A development product from Sun that uses JavaTM technology to greatly simplify the complex process of building business applications that can access any database. Java Blend contains a tool and a runtime environment that simplifies the integration of Java objects with enterprise data.

Java Enterprise APIs
This offers connectivity to enterprise databases via the JDBC package, and to distributed objects with Java IDL interface to CORBA and via native JavaBeans model.

Java IDL
This API provides support for defining remote CORBA interfaces in the IDL interface definition language, an industry standard by OMG. Java IDL includes an IDL-to-Java compiler and a lightweight ORB that supports IIOP.

JDBC (Java Data Base Connection)
A set of interfaces (Java methods and constants) in the Java 1.1 enterprise framework, defining a uniform access to relational databases. JDBC calls from a client or server Java program link to a particular "driver" that converts these universal database access calls (establish a connection, SQL query, etc.) to particular syntax needed to access essentially any significant database.

JFC (Java Foundation Classes)
Core to the Java platform, the Java Foundation Classes extend the original Abstract Windowing Toolkit (AWT) by adding a comprehensive set of graphical user interface class libraries.

Java JDK (Java Development Kit)
Java Development Kit, including Java API for the language foundation classes and the base software development tools such as Java compiler or native class interface generator. Current release is JDK 1.1.5.

Java Media APIs
Support for real-time interactive multimedia including Sound, Animation, 2D, 3D, Java Telephony, Speech and Collaboration.

Java Object Serialization
Allows Java programs to serialize objects into a stream of bytes that can be later used to build equivalent objects for the same or remote JavaVM.

Java RMI (Remote Method Invocation)
Support for creating objects whose methods can be invoked from another virtual machine, analogous to a remote procedure call (RPC). This can be looked upon as a native ORB.

Java Security APIs
A framework for developers to easily and securely include security functionality in their applets and applications. Functionality encompasses including cryptography, digital signatures, encryption and authentication.

JavaSpaces
Distributed persistence and data exchange mechanisms for code written in the Java programming language. Facilitate the development of Systems that use flow of data to implement distributed algorithms.

**Java Servlet APIs**
Support for building customized server solutions in terms of servlets - small executable programs that users upload to run on networks or servers.

**Java Standard Extension APIs**
A standardized framework for extending the core Java API by domain specific packages.

**Java WorkShop**
Java CASE toolkit from SunSoft, includes Visual Java support for visual software engineering.

**JavaOS**
Highly compact operating system designed to run Java applications directly on microprocessors in anything from net computers to pagers.

**JavaScript**
Java-like but typeless and truly interpreted scripted language developed by Netscape and used for dynamic HTML programming both at the client (Navigator) and server (LiverWire) sides.

**JavaVM**
An abstract computer that runs compiled Java programs. Implemented in C as an interpreter of the opcode generated by Java compiler.

**JFactory**
Visual interface builder for Java applications and applets; supports click-and-drag authoring, automated code generation and user level extensibility.

**JNDI (Java Naming and Directory Interface)**
Another product from JavaSoft which allows developers to deliver Java applications with unified access to multiple naming and directory services across the enterprise.

**JSDA**
The Java Shared Data API (JSDA) defines a multipoint data delivery service for use in support of highly interactive, collaborative, multimedia applications.

**Jigsaw**
Web server written entirely in Java, recently published by the WWW Consortium, and to be used as the experimentation platform with mobile code, scripting languages, object-oriented Web etc.

**Just-In-Time Compiler**
A Just in Time Compiler is an Interpreter which "compiles on the fly" by "remembering" the native machine code produced when bytecodes are first seen. This can produce major (at least a factor of 10) improvement in performance of Java Applets when used in a browser and when code is used more than once as it is in any iteration. Symantec produced the first well known JIT for Java under Windows95.

**Lotus Notes**
A major commercial software system suporting aspects of collaboration but at its heart a web-linked document database providing the type of tools also seen in systems such as Cold Fusion for producing with high level tools general Web Interfaces to database material.

**MBONE**
Multicast Backbone, a virtual network based on IETF specification and used by several hundred researchers for developing protocols and applications for group communication on the Internet.

**Microsoft ActiveVRML**
Microsoft proposal for VRML 2.0 that lost with the Moving Worlds proposal from SGI and partners but is being nevertheless implemented by Microsoft.
Microsoft ActiveX
Previous OLE framework, now reformulated as a content development and media integration environment for the Internet. Includes tools for visual programming and integration.

Microsoft ActiveX Conferencing
A suite of technologies that enable real-time, multiparty, multimedia communication over the Internet.

Microsoft ActiveX Controls
Formerly known as OLE controls, are components (or objects) that can act in a similar way as Netscape plug-ins or Java applets in the Microsoft software/database domain

Microsoft ActiveX Development Kit
Includes an evolving collection of technologies that Microsoft is experimenting with and introducing to facilitate development of Internet applications and content.

Microsoft Jakarta
Codename for JavaVM for Windows and visual Java development tools from Microsoft

MPI (Message Passing Interface)
The parallel programming community organized an effort to standardize the communication subroutine libraries used for programming on massively parallel computers such as Intel's Paragon, Cray's T3D, as well as networks of workstations. MPI not only unifies within a common framework programs written in a variety of exiting (and currently incompatible) parallel languages but allows for future portability of programs between machines.

MIME (Multipurpose Internet Mail Extension)
The format used in sending multimedia messages between Web Clients and Servers that is borrowed from that defined for electronic mail.

Netscape Client Pull
Netscape extension of HTML which allows servers to put the Navigator in an active pull mode, i.e. to instruct the Navigator to reload an indicated URL after a specified timeout

Netscape Community System
Web based community network environment, including support for bulletin boards, real-time chat sessions, electronic mail, and private discussion groups.

Netscape Communications Server
High-performance server software that enables organizations to publish rich hypermedia documents (standard Netscape Web server before the current series of new server products)

Netscape Cookies
A "cookie" is a small piece of information which a web server (via a CGI script) can store with a web browser and later read back from that browser. This is useful for having the browser remember some specific information across several pages

Netscape Enterprise Server
High performance, secure WWW server for HTML delivery, content management and portable access to database management systems

Netscape JRI
An abstract interface to Java services that allows application programs (Java service consumers) to be decoupled from the Java runtime internals (Java service producers)

Netscape LDAP
The Lightweight Directory Access Protocol (LDAP) is Netscape's strategic directory protocol. It defines a simple mechanism for Internet clients to query and manage an arbitrary database of hierarchical attribute/value pairs over a TCP/IP connection
**NCSA Habanero**
A framework for constructing distributed collaborative software tools. Using Habanero, programs that were designed for a single-user can be relatively easily recast as multi-user collaborative tools.

**object reference**
A construct containing the information needed to specify an object within an ORB. An object reference is used in method invocations to locate a CORBA object. Object references are the CORBA object equivalent to programming language-specific object pointers. They may be obtained from a factory object or from the Naming Service.

**Object Web**
The evolving systems software middleware infrastructure gotten by merging CORBA with Java. Correspondingly merging CORBA with Javabeans gives Object Web ComponentWare. This is expected to compete with the COM/ActiveX architecture from Microsoft.

**OMG (Object Management Group)**
OMG is the organization of over 700 companies that is developing CORBA through a process of call for proposals and development of consensus standards.

**ORB (Object Request Broker)**
Used in both clients and servers in CORBA to enable the remote access to objects. The ORB provides the basic communication channel through which all objects interact to provide system services. Since all object behaviour is specified in terms of messages exchanged among objects, the communication protocol defined by the ORB is in effect the grammar for all the other OMA specifications. ORB's are available from many vendors and communicate via the IIOP protocol.

**RTP**
A thin protocol providing support for applications with real-time properties, including timing reconstruction, loss detection, security and content identification.

**sandbox**
Java applets run in a womb provided by the web browser that offers them services, and prevents them from doing anything naughty such as file i/o or talking to strangers (servers other than the one the applet was loaded from). The analogy of applets being like children lead to calling the environment they run in the "sandbox". See womb, applet.

**Server**
This is usually a program which listens to a specific port, and accepts (processes) invocations from objects at remote locations.

**server skeleton**
A public abstract class generated by the idltojava compiler that provides the ORB with information it needs in dispatching method invocations to the servant object(s). A server skeleton, like a client stub, is specific to the IDL interface from which it is generated. A server skeleton is the server side analog to a client stub, and these two classes are used by ORBs in static invocation.

**Servlet**
A component of the Java Web Server (formerly known as Jeeves) that facilitates creation of executable programs on the server side of an HTTP connection.

**SSL**
Secure Socket Layer developed by Netscape and used as their base security enforcing technology

**Socket**
Tool for network communications, specifies end-points of a communication. Sockets identify the origin and destination of messages.

**SQL**
Structured Query Language. The language most commonly used to create database queries.
**stringified object reference**
An object reference that has been converted to a string so that it may be stored on disk in a text file (or stored in some other manner). Such strings should be treated as opaque because they are ORB-implementation independent. Standard object_to_string and string_to_object methods on org.omg.CORBA.Object make stringified references available to all CORBA Objects.

**Synchronized**
Java's Abstraction of a mutual exclusion lock. This prevents multiple threads from accessing the same piece of data, thus avoiding deadlocks.

**Televirtual**
The ultimate computer illusion where the user is fully integrated into a simulated environment and so can interact naturally with fellow users distributed around the globe.

**TCP (Transmission Control Protocol)**
A connection-oriented protocol used in the DARPA Internet. TCP provides for reliable transfer of data, as well as out-of-band indication of urgent data.

**VIC**
Public domain real-time Internet video conferencing tool from LBNL

**Visual Java**
Visual Java authoring support in the recent release of Java Workshop, a Java CASE toolkit from SunSoft

**VRML 1.0**
A standard network protocol for 3D scene description based on a subset of the ASCII Open Inventor format

**VRML 2.0**
Extends VRML 1.0 towards interactive 3D worlds by adding Sensor nodes that detect user events, routes that connect user events with the scene graph nodes, and Script nodes that describe interactive behavior of the scene graph objects using a scripting language such as Java, JavaScript, Tcl etc.

**VRMLScript**
Custom script for VRML 2.0 under consideration by the VRML forum

**Web Client**
Originally web clients displayed HTML and related pages but now they support Java Applets that can be programmed to give web clients the necessary capabilities to support general enterprise computing. The support of signed applets in recent browsers has removed crude security restrictions, which handicapped previous use of applets.

**WebCORBA**
Recent initiative by W3C to integrate CORBA with the Web

**WebObjects**
Web server extension technology from NeXT, including WebScript scripting language, a collection of productivity tools, and interfaces to database management systems.

**Web Servers**
Originally Web Servers supported HTTP requests for information - basically HTML pages but included the invocation of general server side programs using the very simple but arcane CGI - Common Gateway Interface. A new generation of Java servers have enhanced capabilities including server side Java program enhancements (Servlets) and support of stateful permanent communication channels. See Java Servers, CGI.
Bibliography


   Daniel Dias, Geoffrey Fox, Wojtek Furmanski, Vishal Mehra, Balaji Natarajan, H.Timucin Ozdemir, Shrideep Pallickara.

4. **Prototype of a Scalable Tele-Virtual Environment on the Web Using VRML2.0 / JSDA IBM T. J. Watson Research Center, April 1997**
   Wojtek Furmanski, Daniel Dias, Balaji Natrajan, Vishal Mehra, Shrideep Pallickara.
   Joint Project with IBM-Watson Research Center & Syracuse University -
   **Principal Investigator:** Dr. Wojtek Furmanski.
   **Included as a part of JSDA distribution by Javasoft.**
   **This paper has also been listed among the interesting works in JSDA at the Java Developers Connection (JDC) site by Javasoft.**

5. **Comparison Of Collaboration Environments A Survey Paper which compares Habanero, JSDA & Shaking Hands**
   Geoffrey Fox, Wojtek Furmanski, Vishal Mehra, Shrideep Pallickara.

6. **Locales and Beacons: Efficient and Precise support for Large Multi-User Environments**
   John W. Barrus, Richard C. Waters, David B. Anderson at MERL Research Labs.

7. **White Paper, Component-Based Software with Java Beans and ActiveX**

8. **IIOP: OMG's Internet Inter-ORB Protocol: A Brief Description**
   Mike Bradley of Object Management Group, May 97.

10. DCOM and CORBA - Side by Side, Step by Step, and Layer by Layer


20. Jang Ho Lee, Atul Prakash, Trent Jaeger and Gwobaw Wu, Supporting Multi-User,


VITA

NAME OF AUTHOR: Shrideep B Pallickara

PLACE OF BIRTH: Bombay, India

DATE OF BIRTH: June 7, 1972

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

BE (Electronics & Telecommunications) Victoria Jubilee Technical Institute (VJTI), Bombay University, India.

MS (Computer Engineering) Syracuse University, Syracuse.

PROFESSIONAL EXPERIENCE:

Graduate Research Assistant, Northeast Parallel Architectures Center, Syracuse University, 1996-98

Java Collaboration Developer – IBM T.J. Watson Research Center 1997
Systrems Analyst – Mahindra-British Telecom 1994-95