Pellucid: an environment for distributed applications
by Marcus Giese

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science
Montana State University
© Copyright by Marcus Giese (2000)

Abstract:
The development of distributed applications for scientific or other purposes has always required extensive knowledge not only of the application domain, but also of the underlying technical infrastructure of the computer systems and software that is used to distribute the applications. Pellucid presents a system based on the Common Object Request Broker Architecture (CORBA) [8, 1] and the JavaBeans [6, 7, 3] event model that requires only minimal infrastructure and support knowledge of the scientist that wishes to develop a distributed system. The components can be used with any JavaBeans compatible, free or commercial development environment, and it only employs distributed technology that is available free of charge. All components are highly platform portable and the environment in which they execute does not have to be homogeneous. In this Pellucid presents both a very powerful, but yet simple model for the development of distributed systems, primarily of scientific applications. The potential for Pellucid is demonstrated by sample applications consisting of components for matrix manipulation.
PELLUCID: AN ENVIRONMENT FOR
DISTRIBUTED APPLICATIONS

by

Marcus Giese

A thesis submitted in partial fulfillment
of the requirements for the degree
of
Master of Science
in
Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

July 2000
APPROVAL

of a thesis submitted by

Marcus Giese

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

Dr. Gary Harkin  
(Signature)  7/24/00  
Approved for the Department of Computer Science

Dr. Denbigh Starkey  
(Signature)  7/24/00  
Approved for the College of Graduate Studies

Dr. Bruce McLeod  
(Signature)  7-24-00
STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library.

If I have indicated my intention to copyright this thesis by including a copyright notice page, copying is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U. S. Copyright Law. Requests for permission for extended quotation from or reproduction of this thesis in whole or in parts may be granted only by the copyright holder.

Signature

Date 7/24/2000
# TABLE OF CONTENTS

LIST OF FIGURES ...................................................................................................... v

1. EARLIER WORK ................................................................................................. 1

2. PELLUCID .............................................................................................................. 4

3. METHODS ............................................................................................................. 9
   USER INTERFACE .............................................................................................. 9
   IMPLEMENTATION DETAILS ............................................................................ 10
   Client Infrastructure ......................................................................................... 12
   Operation Of The Server .................................................................................. 14
   Returning Results ............................................................................................ 16

4. CONCLUSIONS ................................................................................................... 18
   THE ROLE OF PELLUCID ................................................................................ 18
   PERFORMANCE AND OTHER CONSIDERATIONS ........................................ 18
   FURTHER WORK .............................................................................................. 20
   SECURITY ........................................................................................................ 21
   CONCLUSION ................................................................................................. 23

REFERENCES CITED ............................................................................................... 24

APPENDICES ............................................................................................................. 26

APPENDIX A - SOURCE ...................................................................................... 27
APPENDIX B - INTERFACE DEFINITION ............................................................ 29
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Simple Task Graph</td>
<td>7</td>
</tr>
<tr>
<td>2. Simple Matrix Application in Sun BeanBox</td>
<td>10</td>
</tr>
<tr>
<td>3. Simple Matrix Application in Sun AppletViewer</td>
<td>11</td>
</tr>
<tr>
<td>4. High-Level Pellucid Architecture</td>
<td>12</td>
</tr>
<tr>
<td>5. High-Level Event Sequence</td>
<td>17</td>
</tr>
</tbody>
</table>
ABSTRACT

The development of distributed applications for scientific or other purposes has always required extensive knowledge not only of the application domain, but also of the underlying technical infrastructure of the computer systems and software that is used to distribute the applications. Pellucid presents a system based on the Common Object Request Broker Architecture (CORBA) [8, 1] and the JavaBeans [6, 7, 3] event model that requires only minimal infrastructure and support knowledge of the scientist that wishes to develop a distributed system. The components can be used with any JavaBeans compatible, free or commercial development environment, and it only employs distributed technology that is available free of charge. All components are highly platform portable and the environment in which they execute does not have to be homogeneous. In this Pellucid presents both a very powerful, but yet simple model for the development of distributed systems, primarily of scientific applications. The potential for Pellucid is demonstrated by sample applications consisting of components for matrix manipulation.
CHAPTER 1

EARLIER WORK

The development of distributed applications for scientific or other purposes has always required extensive knowledge not only of the application domain, but also of the underlying technical infrastructure of the computer systems and software that is used to distribute the applications. Existing distributed computing environments require the scientist or other potential user to learn a large set of library calls to properly distribute data and synchronize execution. In addition, these environments do not work well with existing Graphical Integrated Development Environments (IDE) such as the Microsoft VisualStudio suite or one of the many Java GUI design environments.

It is nearly impossible to list all prior work and all more or less successful attempts at distributed computing. There are some very successful systems and some that are not very widely used. In the following we briefly examine some of the more well known systems. In addition we describe an earlier attempt by the author to create Pellucid using a Java Jini and JavaSpaces environment as the underlying distributed infrastructure and distribution mechanism.

In general earlier efforts can be categorized into parallel computing environments and distributed environments. For the purpose of this thesis we will focus only on those that fall into the latter group, though some of the parallel environments such as Trapper [18] may offer interesting ideas that could be used to expand Pellucid's
functionality. Also not discussed are Web-based parallel management and monitoring tools such as Conspector [19].

The most widely known and used distributed computing environments are Message Passing Interface (MPI) [23] and Parallel Virtual Machine (PVM) [26]. PVM is a distributed computing environment that allows applications to be developed and distributed across a heterogeneous compute environment at runtime. PVM consists of a set of libraries that can be used to develop distributed applications in several languages. Several add-ons exist that provide features such as load balancing. Using PVM requires that the application developer have a high degree of programming expertise. To the author's knowledge no graphical design tools exist for use with PVM. There are however graphical PVM tools for debugging and monitoring the distributed application.

MPI is a message passing library that can be used by application developers to create distributed applications. As with PVM many graphical tools exist for the management and monitoring of distributed MPI applications [25], but it also requires a high level of programming expertise.

More recent distributed application environments include JavaParty [15] and Parallel Java [27], but again, the level of programming expertise required is substantial.

Many of the existing efforts implement their own load-balancing and management efforts whereas Pellucid uses an industry-standard environment allowing the use of existing infrastructures and tools. For example some CORBA implementations provide load-balancing and fail-over mechanism that Pellucid can automatically
take advantage of. The same is true for the debugging and monitoring of executing applications.

Aside from the non-standard architecture and infrastructure designs, all of these environments assume that the user has a fairly high level of programming experience and understanding of the technical aspects of distributed programming. Pellucid on the other hand offers a true separation of the domain knowledge from the technical and infrastructure knowledge, assuming that the user has a solid understanding of the problem he or she is attempting to solve, but relieving him or her from knowing about the actual distribution and programming details. Using Pellucid to develop requires no programming skills past the ability to use one of the many JavaBeans-enabled Graphical Integrated Development Environments such as Symantec VisualCafe or Inprise JBuilder.

The author developed a version of Pellucid using Sun Jini [11, 10, 28] and JavaSpaces [4] but the reference implementation of Jini provided by Sun Microsystems proved to be too unreliable to be used for a working system. A solution based on Jini and JavaSpaces is attractive because it offers a standard-compliant, Java-only or pure Java option, which also provides a highly fault-tolerant and persistent dataspace with JavaSpaces.
PELLUCID

Pellucid presents a system based on the Common Object Request Broker Architecture and the JavaBeans event model that requires only minimal infrastructure and support knowledge of the scientist that wishes to develop a distributed system. The components can be used with any JavaBeans compatible, free or commercial development environment, and only employs distributed technology that is available free of charge. In this, Pellucid presents both a very powerful, but yet simple model for the development of distributed systems, primarily of scientific applications. The author will demonstrate the potential of Pellucid through the use of matrix manipulation components that are used to create a demonstration application for distributed matrix calculations. It should be emphasized that the examples in this paper show some of the infinite number of components and applications that can be developed based on Pellucid. Pellucid provides a general purpose approach for developing components that inherit the ability to dynamically move around a distributed computing infrastructure.

Distributed systems offer the potential of improved performance, increased fault-tolerance, scalability, and cost savings over large and expensive parallel computers. The main problems with distributed applications has been difficulty of designing and implementing them.

Combining a modern distribution component model architecture (CORBA) and
infrastructure with an easy to use and popular complementary event and application model (JavaBeans), writing distributed applications can be simplified. This is made possible by providing often-used operations in a componentized form that is compatible with a de facto object and component standard such as JavaBeans. Every component provided uses inheritance and aggregation to obtain the ability to be distributed and executed on a network of compute nodes. In essence this establishes mobility of fine-grained objects combined with the JavaBeans “click-and-connect” application design model.

In the past Java has sometimes been viewed as slow and not suitable for scientific applications [17, 16], many of which are compute and/or I/O intensive. Advances in the Java compilers and the Java virtual machines over the last year have made the performance issue much less of a problem [24, 20]. The ability to easily distribute applications over a wide array of available compute platforms can make up for the potential shortcomings in the performance of individual members of this compute environment.

Pellucid implements a system of distributed JavaBeans components that allow matrix manipulations. The basic architecture and design allows scalar datatypes, one dimensional arrays, as well as two- and three- dimensional matrices of any valid non-primitive Java datatype. Because of Java’s introspection and reflection mechanisms all operations could easily be enhanced to transparently handle any other datatype.

The arithmetic operations are built on top of a CORBA infrastructure that allows every component to move itself to any network computer that is running a Pellucid server or that is configured to allow an automatic activation of a Pellucid server. The
server is not specific to any particular arithmetic function or component, but rather
serves as a container to execute any component that follows the Pellucid interface
specifications. In addition to the actual distribution infrastructure, Pellucid compo­
nents are complemented by a mechanism that allows the connection of CORBA events
to JavaBeans events, allowing a seamless integration of the JavaBeans UI design with
the remote CORBA events. This allows a application developer to visually design
the application in a JavaBeans UI environment, selecting JavaBeans components and
visually connecting the components. All components are provided with the name of
a default machine to execute on, but a JavaBean \(^1\) Property Editor for every JavaBean allows the application designer to visually designate components for execution
on specific remote machines.

The following example demonstrates Pellucid’s basic underlying principle. The
matrix expression:

\[
\text{result} = ((A + B) + (B \times C)) \times ((B \times C) \times (C - D))
\]

(2.1)

can be described with the following task graph, shown in Figure 1.

Every node in the task graph is implemented by Pellucid using a corresponding
JavaBean object and each object can execute independently on a different machine
in the network.

Pellucid operations use Java Introspection and Reflection combined with poly­

\(^1\) A JavaBean is a Java class that follows certain naming conventions and patterns
operations supported by the example applications used to demonstrate the Pellucid approach and architecture are:

- Matrix Multiplication
- Matrix Addition
- Matrix Subtraction
- Matrix Inversion
- Identity Matrix
- Input
- Output

All operations are built on top of Pellucid infrastructure support functions that provide communication abilities, fail-over support, execution monitoring and load and node management. Pellucid implements some of these services directly while others are leveraged from the CORBA infrastructure. Support for communication
and transfer of executable code is implemented by Pellucid components, using low-level CORBA marshalling functions and interfaces. Event notification and mapping is implemented by Pellucid using the CORBA call-back mechanisms and load-balancing and fail-over support is only available to the extent supported by the underlying CORBA ORB implementation that is used to deploy Pellucid-based applications, and will differ between CORBA implementations.

Pellucid is unique in its use of the JavaBeans event model to enforce proper operation sequencing and synchronization. As will be briefly explained below, no node in the task graph of shown in Figure 1 can proceed with its execution until its input has been received from preceding operations. This ensures proper sequencing of operations without requiring the use of complex and cumbersome synchronization calls. Pellucid extends this single-machine event model to a distributed environment.
CHAPTER 3

METHODS

User Interface

When selecting a potential User Interface (UI) architecture many possibilities exist. Simplicity and ultimate portability are desirable goals for Pellucid for that reason Java, and more specifically JavaBeans, was chosen for the User Interface implementation of the components. All UI portions of the arithmetic operations were implemented as visual JavaBeans painted in a basic AWT Applet [21]. Figure 2 shows a simple matrix manipulation application created using Sun Microsystems' BeanBox that is distributed with the JavaBeans development kit [29].

Figure 3 shows the same application after generating and running the application as a Java Applet. The applet was generated by the BeanBox using only default values. This is just one of the unlimited number of applications that can be created using the Pellucid distributed architecture. The example shown implements the linear equation shown in Figure 1.

In the left part of Figure 2 the various components that are available for creating an application are shown. Simply selecting a component from the pallet and moving it to the design canvas adds that component to the application. Components can be connected through the Edit menu of the BeanBox which allows the application designer to select from the list of operations a component supports and connect these operations to those of other components on the canvas. The Input and arithmetic
components are Pellucid specific, and the Start button is a standard component available in the BeanBox. Components such as the Input bean have additional controls such as file selection box and fields for specifying the size of the input matrices.

The lower right corner of Figure 2 shows the Property Editor for the selected Add component. Visible is the computer name that the component should execute on (viper). The machine name can be changed just as easily as any of the other properties, the color of the component for example.
Figure 3. Sun Applet Viewer.
Computer I
- Server
- ActivatorImplementation
- AddImplementation
- MultiplyImplementation

Component Implementations
(As Java Byte Code Stream)
& Results

Computer II
- Server
- ActivatorImplementation
- AddImplementation
- MultiplyImplementation

Component Implementations
(As Java Byte Code Stream)
& Results

Computer III
- Server
- ActivatorImplementation
- AddImplementation
- MultiplyImplementation

Component Implementations
(As Java Byte Code Stream)
& Results

CORBA Object Request Broker (Object Bus)

Component Implementations
(As Java Byte Code Stream)
& Results

JavaBeans IDE
- AddImplementation
- MultiplyImplementation
- LocalImplementation

Client Desktop

Figure 4. Pellucid Architecture.

Implementation Details

Figure 4 shows the high-level Pellucid architecture and its primary components.
Client Infrastructure

The DistributedObject class is the most visible part of the Pellucid architecture. This class is a direct sub-class of Object and implements the interfaces Runnable, Serializable, and UniObject. Of these, only UniObject is a new Pellucid interface and it defines the minimum operations the application or compute component must implement and support when it is executing remotely. These four operations are Start, Stop, Suspend, and Resume. The other two interfaces, Runnable and Serializable guarantee that every Pellucid component can be invoked in a thread on the remote machine, and that the component can be serialized to be transported to a remote execution node, respectively.

DistributedObject's most important operation is Activate. Activate has two primary responsibilities. The first task is to create and set up a CORBA [22] connection to a remote execution machine. This is accomplished through Pellucid's SimpleCORBA class which will be described in more detail later. When DistributedObject creates the CORBA connection and initializes the Object Request Broker environment, it also passes along a reference to the subclass object's JavaBeans input listener. For example, if the requesting object is a matrix multiplication node, a reference to this component's input listener is passed to the SimpleCORBA instance. This is important, as it is the LocalImplementation instance created by SimpleCORBA that will receive the remote CORBA event when the remotely executing object has finished execution and is ready to return the results of the computations. DistributedObject's second responsibility is to invoke the Activate method on the SimpleCORBA object.

\(^1\)in the object oriented sense
it created. This causes SimpleCORBA to initiate the actual network transfer and invocation of the component that is to be executed remotely.

When a SimpleCORBA object is created by DistributedObject the constructor initializes the Object Request Broker (ORB) environment, configures the communications protocol to Internet Inter-Object Request Broker Protocol (IIOP), and binds to the remote Pellucid server’s Activator interface. After binding to the remote interface SimpleCORBA creates an instance of of the Pellucid LocalImplementation class, adds the subclassed object’s inputlistener to it, and registers it with the remote Pellucid activation server as a callback. When the remotely executing object has finished, the Pellucid server will invoke this callback’s handleInput() method and pass it the result object it received from the remote component.

After SimpleCORBA has initialized the ORB environment and registered the distributed component’s callbacks with an instance of LocalImplementation and the remote Pellucid activation server, its ActivateObject method can be called by DistributedObject. ActivateObject serializes the distributed component converting it into a Java byte array which is recognized by the CORBA infrastructure as a CORBA octet sequence, as defined in the interface file Activator.idl. It then creates a new Java thread to send the byte stream to the remote Pellucid server as shown in Figure 4. No further action on the part of the client or originating machine is required; as soon as the byte stream arrives at the remote machine, it will be instantiated, and its Start() method will be invoked, commencing remote execution of the distributed component.
Operation Of The Server

The Pellucid server consists of two components. The first part is a main server whose two tasks consist of registering the Pellucid Activator with the host’s Object Request Broker and creating an instance of the actual Activator. After Server has registered the Activator interface, and created the Activator instance, it remains inactive until the Pellucid activator service is shut down.

The second component of the Pellucid server is the Activator, which performs the following tasks:

- receive the serialized distributed objects
- instantiate the objects
- accept callback registrations for the received objects
- invoke the Start() method of the instantiated, distributed objects to begin the computational work of the object
- on the behalf of the distributed object, invoke the client’s callback method (handleInput() in LocalImplementation) to return the computation’s result when the distributed object’s Thread.run() method has finished its work
- destroy the object

The ActivatorImplementation's ActivateObject() method receives the serialized component from the client and instantiates the byte array into the appropriate object. The actual type of the received object is immaterial as long as it implements the Pellucid UniObject interface. The activator then inserts a reference to the instantiated object into an internal hashtable in order to be able to later retrieve information specific to a particular object, since one activator can instantiate many different
components. At this point the activator invokes the \textit{Start()} method on the newly created component instance, as defined by the \textit{UniObject} interface, which in turn starts a new thread executing the \textit{run()} method of the transmitted, distributed object. Once this \textit{run()} method has finished its computations, it can return the result to the client.

Returning Results

When a component's \textit{run()} method has finished its work and the component is ready to return its results, it invokes its \textit{returnResult()} operation. Every distributed Pellucid component inherits this operation from \textit{DistributedObject}. The \textit{DistributedObject returnResult()} method then invokes the \textit{returnResult()} method on the activator instance.

The \textit{returnResult()} method built into the Pellucid \textit{ActivatorImplementation} serializes the result passed to it by the distributed component executing locally to the server, and invokes the \textit{handleInput()} method on the \textit{LocalImplementation} located on the client. This is the callback the \textit{SimpleCORBA} instance registered before activating the remote component. When the \textit{LocalImplementation} receives the result, it deserializes it, and sends the result to all local class instances that registered an \textit{InputListener} JavaBeans event for it. At this point the local JavaBeans event structure and model resumes its regular operation and computation proceeds through the JavaBeans taskgraph, potentially starting the cycle of component distribution over with operations that are defined at later points in the taskgraph, i.e. operations that are dependent on earlier results.
Figure 5 shows a simplified diagram of the sequence of events for an example *Add* component.
CHAPTER 4

CONCLUSIONS

The Role Of Pellucid

The role of Pellucid is not to replace any other distributed application environments, but rather to provide an option that allows the fast transformation of scientific computational problems into working distributed applications. Emphasis has been placed on using industry-standard off-the-shelf components (COTS) and on making it possible to develop these applications from within one of the many popular Graphical Integrated Development Environments, many of which are free of charge at least for the entry-level versions.

Because the emphasis has been placed on ease of use and accessibility, some trade-offs had to be made. Pellucid is not a truly general purpose distributed compute environment, though its existing components can be complemented and extended by many freely available JavaBeans components that allow the inclusions of not only a large number of graphical components or widgets, but also include wiring and logic JavaBeans, which are available. Using these and perhaps more components and operations written to the Pellucid interface, the current environment can be made general purpose.
Performance And Other Considerations

Pellucid’s distribution mechanism has not been optimized for execution performance. For example, entire objects are transferred for every object execution even though only the data portion really needs to be transmitted every time. This could be addressed by separating the data from the code in a non-object-oriented fashion and then only transmitting the code if the code has changed. However, the ability to transmit all parts of the object, including the actual class file bytecode, is a potentially useful feature that could be further extended to improve Pellucid’s functionality, as remote compute nodes would not need prior knowledge of the component’s class definitions. This could even be user-configurable.

Furthermore, no effort was made to optimize the CORBA environment in any way. For the current implementation Iona’s OrbixWeb3.1c [2] was used and OrbixWeb does have various optimization parameters for object activation and data transmission, as do most other free and commercial CORBA Object Request Brokers.

Execution performance of the distributed components is limited by the performance of the remote Java Virtual Machines (JVM). While JVM performance has significantly improved, it still lags behind native C or C++ code for raw execution speed. In addition JVM performance varies widely from vendor to vendor and between different operating system platforms.

Load-balancing for the components can be handled by the CORBA Object Request Broker environment, which makes it possible to distribute object activation
requests over a cluster of compute nodes based on their relative performance characteristics and system loads. The most significant restriction in this regard lies with the performance of the controlling machine. This is the computer that originated the computation and that serves as a event controller, i.e. the machine whose Java Virtual Machine executes the task graph of events created by the application.

Compute node failures can be handled by the CORBA ORB environment, with the exception of a failure of the controlling machine. Failure of the controlling node results in failure of the computation. It may be possible to mitigate this risk by making use of persistent CORBA objects which can be used to check-point the computation and restart it after a failure. It is also possible to employ other mechanisms, such as redundant Java Virtual Machines and object cloning to provide a backup and fail-over option for the controlling node, though this has not been implemented in the current version of Pellucid.

Further Work

In order to further improve Pellucid's useful application range several things can be done. It would be very beneficial if the granularity of Pellucid's operations could be improved. For example some matrix operations, such as the multiply, could be broken down further into vector operations which could then be distributed using Pellucid's distribution mechanisms. Also, currently logical operations such as comparison and decision statements are limited to a single node because no Pellucid specific logic JavaBeans have been developed. This is an area in which further work is warranted.

The environment could be made more flexible and usable if all transferred classes
or new component classes were created as true CORBA components that could be stored in and advertised from a CORBA Interface Repository (IR). A CORBA IR can be accessed from CORBA compliant applications, written in any of the languages that have a CORBA mapping. This would allow even non-Java applications to invoke distributed Pellucid components and incorporate their compute capabilities into a larger class of applications.

Currently Pellucid’s single most significant shortcoming is its single point of failure for the control mechanism. While individual distributed components can be made redundant through CORBA Object Request Broker features, the originating machine still poses a single point of failure. If the machine that originated the application fails, the entire computation fails because the remaining components do not have an independent peer-to-peer communication and control mechanism.

Pellucid would benefit from an error handling system. Currently the only errors that are handled are system, communication, or other infrastructure errors. Application errors such as an attempt to invert a singular matrix are not handled at all. Pellucid’s architecture however makes it possible to add this functionality to components and applications based on the Pellucid model.

Security

Pellucid currently has no security mechanism that would provide encryption, data security, or authentication. Depending on the environment Pellucid operates in, this could be a serious issue, being less of a problem in an academic environment than in a commercial setting. In the current implementation it is possible for an unauthorized
client to submit objects for execution to a server. It is also possible for a server to make unauthorized modifications to the object's data and thus return no or even worse, incorrect results. The server is also not very well protected against malfunctioning or malicious components. If we assume that the server machines can be kept secure, then we can solve the problem of unauthorized client invocations by requiring authentication based on public key encryption and signatures. Unfortunately this may not be a realistic scenario if we want to take advantage of the untapped compute power of large PC farms. The problem of malicious servers returning falsified data or results is most likely more serious and should be addressed in any production system based on the Pellucid architecture. The challenge is the fact that the server PCs cannot be trusted, i.e. the owner or user of the PC at a minimum will have sufficient access to start up rogue servers. One possible solution to the problem could be the dynamic downloading of the "real" execution server. In order for this to work every PC would run a temporary server whose sole purpose would be to download the real server. When execution of distributed applications is supposed to commence (say after 19:00), a trusted and secure server will contact all of the PC servers and upload the "server of the day". Keeping in mind that the real server itself is just a simple program written in Java, it would be possible to encode a signature or CRC, based on either public or secret keys that would expire at the end of the compute cycle (say 06:00), much like a Kerberos ticket. Now every client could retrieve the "key of the day" from the secure server, and use this key to verify that it is talking to and executing on a legitimate server, i.e. a server that is not going to maliciously
modify its data. To keep any other malicious program running on the PC servers from modifying the machines memory directly, all object data in memory could be encrypted. If someone modifies the temporary, download server, the worst that will happen is that the real server will not be downloaded at all.

Conclusion

In summary, Pellucid’s main strengths are its ability to distribute (remotely execute) any of its supported operations, its ability to ensure proper synchronization of operations without using complex library calls, and its ability to be used in any of the many free or commercial Java Graphical User Interface environments and editors in which every node of an applications task graph can be represented by a Pellucid JavaBean operation. For example, this allows a user to leverage the Pellucid environment to perform matrix operations similar to those supported by a tool such as Matlab, with the added benefit that the computations can be distributed over a number of computers. In addition, it is possible to easily extend Pellucid’s operation through the addition of new JavaBeans, implementing new operations and functionality. It is also possible to mix many non-Pellucid third-party JavaBeans with Pellucid’s operations as long as the new operations do not need to be distributed.
REFERENCES CITED

Java and CORBA: Client/Server Programming. 1998

OrbixWeb: Programmer Reference. 1998


JavaSpaces: Principles, Patterns, and Practice. 1999

Java Programming: with CORBA. 1997

JavaBeans: by Example. 1997


Java: Distributed Computing. 1998

Jini in a Nutshell: A Desktop Quick Reference. 2000

Core Jini: Java Series. 1999

Distributed Algorithms. 1996

An Introduction to Distributed Algorithms. 1996


"http://wwwipd.ira.uka.de/JavaParty/"

[16] WWW URL: Sun Java Performance.
"http://sunsite.ee/java/whitepaper/java-whitepaper-10.html"

[18] WWW URL: Trapper Web Site.
"http://www.genias.de/products/trapper/trap_index.html"

[19] WWW URL: Conspector Discussion

[20] WWW URL: IDG Java Performance
"http://www.idg.net/crd_java_5857.html"

[21] WWW URL: Sun Microsystems’ Java Web Site
"http://java.sun.com"

[22] WWW URL: Object Management Group Web Site
"http://www.omg.org"

[23] WWW URL: MPI Web Site
"http://www.erc.msstate.edu/labs/hpcl/projects/mpi/"

[24] WWW URL: IBM Java Performance
"http://www.ibm.com/java/education/javahipr.html"

[25] WWW URL: Graphical Tools for Parallel Programming
"http://www.arc.unm.edu/Workshop/tools/tools.html"

[26] WWW URL: PVM Web Site
"http://www.erc.msstate.edu/labs/hpcl/projects/pvm/"

[27] WWW URL: Parallel Java Web Site
"http://roscoe.bu.edu/talks/javacse/parallel-java.html"

[28] WWW URL: Sun Microsystems’ Jini Web Site
"http://www.sun.com/jini"

[29] WWW URL: Sun Microsystems’ Beans Development Kit Web Site
"http://java.sun.com/beans/software/bdk_download.html"
APPENDICES
I have decided to not add unnecessary bulk to this thesis by printing out all of the source code, but the complete source will be available from:

http://www.cs.montana.edu/~giese

as long as my computer account at Montana State University is active.

Included in this document as Appendix B is the Interface definition that all Pel-lucid components are built to.
APPENDIX B

Interface Definition
typedef sequence<octet> JavaObject;

interface Ilocal
{
    void handleInput(in JavaObject objObject, in long rows, in long cols);
}

interface IActivator
{
    oneway void registerInput(in Ilocal callback);

    long ActivateObject(in JavaObject objObject, in string id);

    long ReturnObject(out JavaObject objObject, in string id);

    long ReturnObjectData(out JavaObject objObject, in string id);

    long ReturnObject(out JavaObject objObject, in string id);

    long ReturnObjectData(out JavaObject objObject, in string id);

    // check the status of the object, execution of the remote object
// is not affected, which also means that since all calls are
// asynchronous, and the remote object keeps executing as well, that
// the status is not guaranteed to be current.

long StatusObject(in string id) ;

// stop execution of the remote object, the remote object is stopped
// but not deleted. Use ReturnObject or DeleteObject to delete the
// remote object.

long StopObject(in string id) ;

// temporarily halt execution of the remote object. This allows a
// reliable StatusObject, ReturnObject, or DeleteObject. Use
// ResumeObject to restart the object.

long SuspendObject(in string id) ;

// resumes execution of a remote object that had been suspended by
// SuspendObject

long ResumeObject(in string id) ;

// deletes the remote object. The object must already have been
// stopped by either a StopObject or a ReturnObject in order to
// be deleted, otherwise the delete request will simply be ignored

long DeleteObject(in string id) ;
} ;