CAT: A High Performance, Distributed Component Architecture Toolkit for the Grid

Juan Villacis, Madhusudhan Govindaraju, David Stern, Andrew Whitaker, Fabian Breg, Prafulla Deuskar, Benjamin Temko, Dennis Gannon, Randall Bramley

Department of Computer Science
215 Lindley Hall
Indiana University
Bloomington, IN 47405

Abstract

Grid systems such as Globus, Legion, and Globe provide an infrastructure for implementing metacomputing over the Internet. The Component Architecture Toolkit (CAT) provides a software layer above the Grid that facilitates programming and end user interaction with the Grid.

1. Introduction

Metacomputing systems such as Globus [10], Legion[13], and Globe [19] provide sophisticated service layers which allow users to access and manage distributed hardware resources. These "Grid" systems provide mechanisms for locating and monitoring machines and networks, protocols for security and authentication, mechanisms for remote file access, and tools for creating and scheduling jobs [11]. However, building distributed applications by programming directly to low level Grid APIs is not easy. End users who wish to solve problems using the Grid, tend to first think in terms of higher-level, problem-centric concepts, such as determining which software resources are applicable, and then designing and building an application using those resources. Low-level details such as process instantiation, machine/network characteristics, and so forth typically come at a later stage of the application building process.

The Component Architecture Toolkit (CAT) provides a software layer above the Grid that enables end users to make use of Grid services for building and running applications composed of distributed software components [17]. In the next section, we describe what components are, and how they can be used in the context of the Grid.

2. Overview of Components

Our model of a component is based on the following definition:

> Components are software objects that provide implementations of a set of standard behaviors. These behaviors are defined by the component framework to ensure that components can be composed and interoperate efficiently and without conflict.

The dependences of one component upon another and the communication between components are completely defined by a set of interfaces. In some frameworks these interfaces describe the events that components generate that are responded to by other components. In other frameworks they describe data stream that pass between components. In still others, they describe uses-provides interfaces that specify the methods that a component may call that are be provided by others.

The framework provides a mechanism in which component instances can be created and have their externally editable properties examined and modified. It also provides a mechanism for connecting components, i.e. specifying which component listens for events generated, or provide services that are used, by a particular component instance. In some systems component connections are made during a "component assembly phase" prior to the application execution. In others, connections can be made on-the-fly, i.e. dynamically as the application proceeds.

Components are assembled and connected together within a "container" object provided by the framework. In the case of distributed systems, a proxy object is used to represent the component within the container. In most systems, a container can be also converted into a component allow-

ing a nesting hierarchy of component instances within an application.

Distributed architectures often catalog components that are available at different sites by using a distributed directory service. These services often contain proxies as well as the metadata about components. For example, information about required runtime environments that a component needs, or special information about property editors.

A software component is different from mere aggregation of library codes in a standalone program. Components are meant to be used in more dynamic settings where executions are event-driven and in which few assumptions (beyond what is specified in the interfaces) about the operating environment can be made. Furthermore, components act as indivisable units of computation, where either the execution as a whole goes forward or not, and is independent of how the component operates internally (e.g., in parallel, serially, asynchronously, etc.).

In the context of the Grid, components serve as the basic building blocks for programming distributed applications.

3. Example Systems

There are a number of important component and component-like systems that have been developed. These range from commercial products like DCOM[7], the proposed CORBA component model[15], Enterprise Java Beans[1], AVS[2], Explorer [14] and Khoros[12] to research systems like WebFlow[3], SciRun[18], NetSolve[9], Ninf[16], Infospheres[6] and others. In addition, a DOE sponsored group is defining a "Common Component Architecture" (CCA)[8] for scientific applications. In this section, we describe the relationship of these systems to the CAT.

3.1. The Software Hierarchy

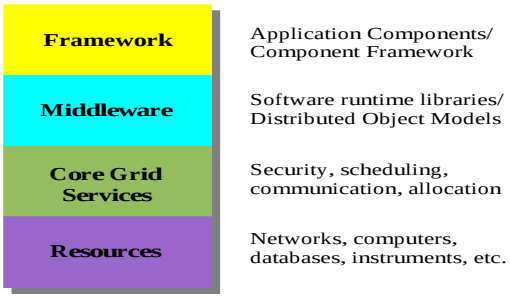

Figure 1. The Software Hierarchy

To understand the relationship between the different component models, object models and Grid systems, it is helpful to consider the software hierarchy in Figure 1. At the bottom level we have the basic Grid resources such as networks, computers, databases, instruments, etc. These resources are accessible through core Grid services such as security, communications, scheduling, allocation, etc. Globus provides the most well known example of this type of grid service system. Legion and Globe are the other prime examples of Grid systems. The difference between these become apparent at the object model level.

The next level up in the hierarchy can be called the object model layer. In addition to providing an inheritance-based object model, this layer also provides location and naming services. The most important examples of grid object models include Legion and Globe. In the case of Legion, it also includes a core grid service layer as part of its design. On the other hand, Globus uses the library approach in which it only provides framework programmers with a set of services that can be accessed, but it does not provide a standard object model. However it is possible to add an object model layer above globus. For example, HPC++ [4] is a distributed object framework, which is largely compatible with Java's RMI distributed object model, and runs over Nexus, the core Globus communication library. HPC++/Java uses the Java naming and RMI registry services.

CORBA from OMG is the most widely known distributed object middleware systems. Although CORBA provides a rich set of services, it does not contain the grid level allocation and scheduling services found in Globus and Legion. On the other hand, it is not difficult to integrate CORBA with either of these Grid systems. In addition it would not be difficult to build the component architecture described in this paper on top of a combination of CORBA and a Grid layer like Globus or legion.

The middleware layer of a distributed system represents the "platform" upon which a component architecture framework is built. The Java Bean component architecture is built on the Java object model and uses an event-listener mechanism to compose components. It is also not a distributed component model. The Enterprise Java Beans (EJB) model is a distributed component model that is also designed to support CORBA client programs. EJB is targeted at the design of server components that communicate with remote clients. The most well known component model is Microsoft's Distributed Component Object Model (DCOM). DCOM is primarily limited to Microsoft Windows, but there are several Unix based implementations. The Object management group has proposed a component model extension to CORBA which is intended to interoperate with EJB. The CORBA proposal has not yet been adopted.

The DOE Common Component Architecture (CCA) is a specification that is based, in part, on ideas contained in the CORBA proposal. Like the CAT model described here, CCA is based on the concept of connecting components by type-compatible "ports". This is not a new idea. It is com-

mon to SciRun from Utah, IRISexplorer, Khoros, AVS, and, to some extent. WebFlow. In addition, the port-component-connection model is used by systems like JavaStudio to connect Java Beans.

4. Motivation

For large scale distributed scientific and engineering applications there are several additional concerns that go beyond those of the standard component model design constraints:

1. A component in a scientific and engineering application may be a computational service that runs on a large, parallel supercomputer. In this case, it is necessary to consider the problem of marshaling and unmarshaling data structures that are distributed. Indeed, the communication protocol between components of this type may consist of parallel streams interleaved over a very high speed network.

2. The volume of data traffic in these components may be very large and special protocols may be required to move it. Can the component communication transport protocol adapt to this situation?

3. Long running applications may require frequent checkpointing of the entire state of all threads of a computation. Java serialization is not sufficient for this task. What level of support is provided for this type of fault recovery?

4. As performance is central to the workings of a scientific application, does the component system provide mechanisms to easily instrument the components?

An overview of how the CAT addresses these issues is given in the following sections.

5. Use in Scientific Computing

In traditional scientific computing, programmers have produced an abundance of standalone libraries which scientists can incorporate into their own codes. This approach, albeit useful, is static and usually requires intimate knowledge about the nature and quirks of the program before it can be fully and safely utilized. That is, these codes are mostly for developers rather than for end users. Thus, these codes may not be used beyond a small circle of elite users.

In contrast, component-style programming can widen the potential user base of such standalone codes. It does this by mandating that all components provide and implement a set of well-defined interfaces that end users as well as programmers/programmatic tools, can use to indirectly access the code. This indirection is implemented as the "component wrapper", and it functions as a layer interfacing the standalone code to the component framework. This allows the framework to treat disparate standalone codes in a uniform way, without requiring special knowledge about the purpose or internal workings of the codes. Moreover, this shifts the "design" phase of program development closer to the "execution" phase since the embedded codes are pre-built/compiled and ready to use. This feature facilitates rapid application development.

The CAT provides a library and runtime environment suitable for component-style programming.

6. Overview of the CAT Framework

The CAT framework is composed of four cooperating systems that provide the following services: component information, component creation, component communication, and component control. Each system is accessed through a well-defined interface by other systems. The CAT Resource Information Service (RIS) provides mechanisms for storing and accessing information relevant to a component's execution. This includes general component information such as the component's purpose and the kinds of inputs and outputs it has, as well as implementation specific information such as what hardware and networks it may be run on. The CAT also provides a generic component process creation facility, dubbed the "procreator", that encapsulates all the polices, usage restrictions, job scheduling, etc. that are involved in starting up processes on host machines. Once a component is started, it is able to communicate with other components through a special form of remote method invocation (RMI). The CAT uses RMI to automate and manage the execution of components, as well as to enable users to control these components remotely.

7. The CAT Component Model

The component model the CAT presents to the user is one based on the notion of "ports". Ports are interfaces which can be thought of as generalizations of inputs and outputs associated with a function call. More specifically, a port is a typed communication channel through which components can delegate their external I/O. Components may have any number of input or output ports associated with them. Each input port can be connected with at most one output port; however, output ports may have an arbitrary number of input ports connected to it. Communication between components is established when an output port on one component is connected to a type-compatible input port on another, and data is passed between the ports. Data transfer is initiated when a component writes to its output port

`String getName()`	name of the component given by the component developer
`String getReadme()`	short description of what the component is/does
`String getStateInfo()`	short description of current internal state of the component
`STATUS getStatus()`	current component status, e.g., `BUSY`, `WAITING`, etc.
`RETVAL addListener (StatusListener c)`	add component status listener
`RETVAL removeListener (StatusListener c)`	remove component status listener
`Vector getInputPorts()`	list of current input ports
`Vector getOutputPorts()`	list of current output ports
`RETVAL setSignal(SIGNAL s)`	set a signal
`Vector getStatusList()`	get the current list of stati defined for the component
`Vector getSignalList()`	set the current list of signals defined for the component
`RETVAL setQuery(QUERY q, String args)`	set and evaluate the query
`Vector getQueryList()`	get the current list of queries defined for the component
`RETVAL initialize(String[] args)`	initialization command line arguments to pass to component
`Vector getPostExecutePorts()`	list of output ports involved in the most recent execution
`String getResultsURL()`	URL of where/how to access output content
`Vector getPreExecutePorts()`	list of input ports required to be ”filled” prior to execution

Table 1. The `CAT2component` interface

and notifies each of its input port listeners that new data is available for reading.

In the CAT model, components and their ports are required to implement a set of standard framework interfaces. These interfaces allow the CAT to obtain structural and semantic information about the components and ports, which can be subsequently presented to the end user. The interfaces also provide a control mechanism that allows the objects to be manipulated by the CAT, and indirectly by the end user. Tables 1-4 show the component and port interfaces using an Java-like syntax.

A brief overview of how and when these methods are used is explained in the next section.

Ports simplify the conceptual model for users since they need only be concerned with what a component accepts as input (from other components or from the user) or what it produces as output. The CAT component model shields users from various underlying complexities (e.g., how data

`String getType()`	port type represented as a fully-qualified Java class name
`STATUS getStatus()`	current port status (e.g., `BUSY`, `EMPTY`, etc.)
`String getName()`	name of the port as provided by component developer
`String getReadme()`	short description of the port’s purpose, usage, etc.
`CAT2component getComponent()`	the component asssociated with this port
`RETVAL getTag()`	”meta data” associated with the port
`String getStateInfo()`	current internal state of the port

Table 2. The `CAT2port` base interface extended by input and output ports interfaces

`Object read()`	read from the input port
`RETVAL write(Object d)`	write data `d` to the input port

Table 3. The `CAT2inputPort` interface

`Object read()`	peek at the data contained on the output port
`RETVAL propagate()`	propagate the data to the input port listeners
`RETVAL addListener (InputPort i)`	add input port `i` to list of listeners
`RETVAL removeListener (InputPort i)`	remove input port `i` from list of listeners

Table 4. The `CAT2outputPort` interface

`RETVAL`	generic return value object that indicates success (or not)
`QUERY`	object used to enumerate simple query functions
`STATUS`	object used to store current state information
`SIGNAL`	control signals used to manipulate the component

Table 5. Class types used in CAT framework

transfer/conversion takes place) so that the user can instead focus on higher level application building.

8. Control Model

In the previous section, we described the calls that may be invoked on a component and port by the framework. However, equally important is the *protocol* for using these calls. In this section we describe how the CAT interacts with the component and ports through these interfaces. Note that this protocol is independent of the actual implementation.

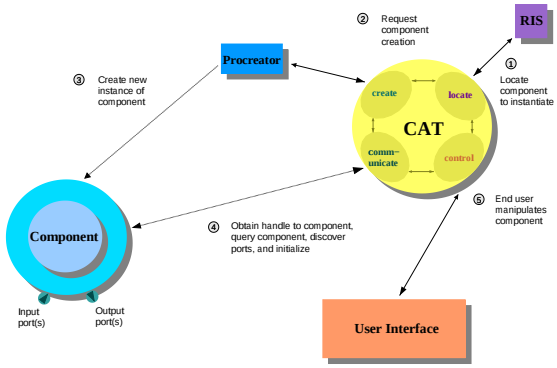

Figure 2. Component location, creation, communication, and control

During a component's initial startup (i.e., after the component has been created), the CAT will invoke the component's `initialize()` method. This method allows a user of the component to pass any initialization information that may be necessary for the component to properly execute. The CAT makes subsequent calls `getName()`, `getReadme()`, `getQueryList()`, `getSignalList()`, and `getStatusList()` to further acquaint itself with the component. The CAT also makes queries to discover the ports associated with the component through `getInputPorts()` and `getOutputPorts()`. Once the CAT has a handle to a port, it can query the port for its type information via the `getType())` method. This type information can be used later to test port compatability between input and output ports (e.g., prior to connecting components via their ports).

Once a component has been started up and initialized, three ways of obtaining component state information become possible. In the first case, a component can determine its own control and notify the CAT of any changes in its state via the `addListener()` method. Here the CAT passively waits for component status updates. In the second case, the CAT assumes a more active role by polling the component through its `getStatus()` method. This is most useful when both the CAT and component share control of the component's execution. In the third case, the CAT directly controls the component's execution and can infer the component state (e.g., busy, modified, waiting, etc.). This latter mechanism of component control is achieved via the `setSignal()` method which provides a hook for the CAT to send various control signals, such as `SIGNAL.EXECUTE`, `SIGNAL.KILL`, `SIGNAL.SUSPEND`, and so forth to the component. Some signals may require pre- or post- processing in order to make use of them. For example, the CAT will invoke `getPreExecutePorts()` to obtain a list of input ports that must have data prior to a component's execution. The CAT can use those ports to notify associated components (and ancestor components if necessary) to forward data to the component(s) which need them. Once a component's execution completes, the CAT invokes a `getPostExecutePorts()` to obtain a list of input ports which were affected by that execution. This list is subsequently used to issue further `setSignal(SIGNAL.EXECUTE)` calls on the associated components.

By default, it is assumed that execution flow follows data flow within a set of connected components.

9. Implementation

The CAT software is composed of three parts: a framework library, a set of composition and control tools, and a set of resource information tools. Underlying it all is our choice of Grid: Globus. Globus was chosen since it was the most developed in terms of high-performance communication protocol support (Nexus), authentication and component creation schemes (GSS, GRAM), and resource location mechanisms (MDS). Furthermore, Globus is implemented in several languages (C/C++ and Java) which facilitated higher level tool building (NexusRMI). Legion shared many of these features and a port of CAT to Legion will be undertaken as soon as resources are available.

The framework library constitutes the core logic behind our CAT component model. It has been implemented in both HPC++ [4] (for the component wrappers embedding C/C++/Fortran codes) and Java (for non-compute intensive components and for the CAT workspace controller). The reason for using HPC++ to write most of the backend components was because the performance of its communication library (based on Nexus) scaled much better than the alternative (a purely Java approach) as data transfer size increased.

The composition and control tools are used to visually manipulate components as well as to control their external execution behaviour. These tools were written primarily in Java since Java provides a rich object oriented language and sophisticated set of standard libraries (AWT, network-

ing) that are portable and easy to use. The communication between the Java-based CAT controller and HPC++-based backend components was achieved using NexusRMI[5], an alternative to JavaRMI which uses the Nexus protocol.

The resource information tools are used to access and modify the RIS databases. These databases are implemented in two parts: a distributed LDAP database, and a network of CAT registries. LDAP is a lightweight directory protocol service that stores data (text, binary) entries in a hierarchical (tree-structured) format. It is used to store mostly static information about components, and is optimized more for reading than for writing. We have taken advantage of LDAP's extensibility features to cross-reference our component entries with machine entries stored in the Globus MDS (another LDAP-based database service). The second part of the RIS databases is the network of CAT registries. A CAT registry is a simple table that associates a name with a remote reference (in this case, a reference to a live component). It is built on top of the Java RMI registry. The CAT provides tools that allow a user to publish information about a running component to the LDAP database so that other RIS users may be aware and potentially make use of the services provided by the component.

The graphical user interface for the cat systems is based on a drop-and-drag component composition system. The reader interested in seeing detailed screen-dumps (or obtaining the CAT software), is directed to the website

`http://www.extreme.indiana.edu/cat`

10. Example CAT Session

A typical LSA scenario using the CAT is shown in Figure 3.

In this example, the user has created an component application that attempts to solve a linear system using three strategies. The internal structure of the application is a tree, and follows a simple dataflow model. At the root of the tree is the NewSystem component, which serves as an input source for the application. This component reads in a linear system data file from disk and converts it to a data structure which can be sent to other components. Connected to the NewSystem component are an informational component, BasicInfo, and two filter components, Scale and Reorder. The BasicInfo component was used to determine the properties of the input linear system, which helped guide the user towards selecting effective solution strategies. The first strategy was to simply solve the system using the SuperLU solver. The second strategy involved first scaling the system and then using a sparse iterative solver on it. The user also chose to view the effect that scaling had on the system, and so connected a Visual component to the Scale component. The third strategy was to reorder the system instead, and then use a similar sparse iterative solver on it.

During the process of building the application, the user elected to run components spread across different host machines. Afterwards, the user brought up the Framework Analysis Toolkit (FAT) tools to view the performance statistics of the NewSystem component, as well as an automatically generated web page that logs the execution results for the SuperLU component. These tools helped the user figure out which strategies were more effecient at solving the given system.

11. Real-world Example using the CAT

A prototype of our CAT software was used in a high-performance computing challenge at the SuperComputing'98 conference. We built an application containing linear system solver components distributed across several offsites machines, together with a parallel finite-element component and a 3D visualization component onsite at the convention. This application simulated the filling of an engine block mold which had 2.55 million elements and required 16GBytes of memory to run. Several key HPDC issues were addressed in our demo. First, the underlying engineering problem required developing parallel solution techniques for time-dependent, multi-physics finite-element simulations on complex geometries. Second, certain resources required for the simulation, such as the 3D visualization device and compute nodes for the finite-element solvers, were not all available in one place. Other issues, such as machine-specific software licensing and the proprietary nature of some of the industrial codes required that the component wrappers be built strictly from binary-only libraries and their specifications, and that the resulting components be run only at certain host sites. This showed how any code, whether textual or library based, could potentially be incorporated as a component into our framework in an accomodating way.

12. Conclusion

Taking advantage of the Grid for high-performance scientific computing requires a software framework layered above the Grid that supports a new style of programming: component-based programming. The CAT facilitates such programming by providing a conceptually simple "port-based" component model together with a suite of developer tools for incorporating existing scientific codes into the framework, as well as a set of end user tools for locating, composing, building, and running distributed component applications over the Grid.

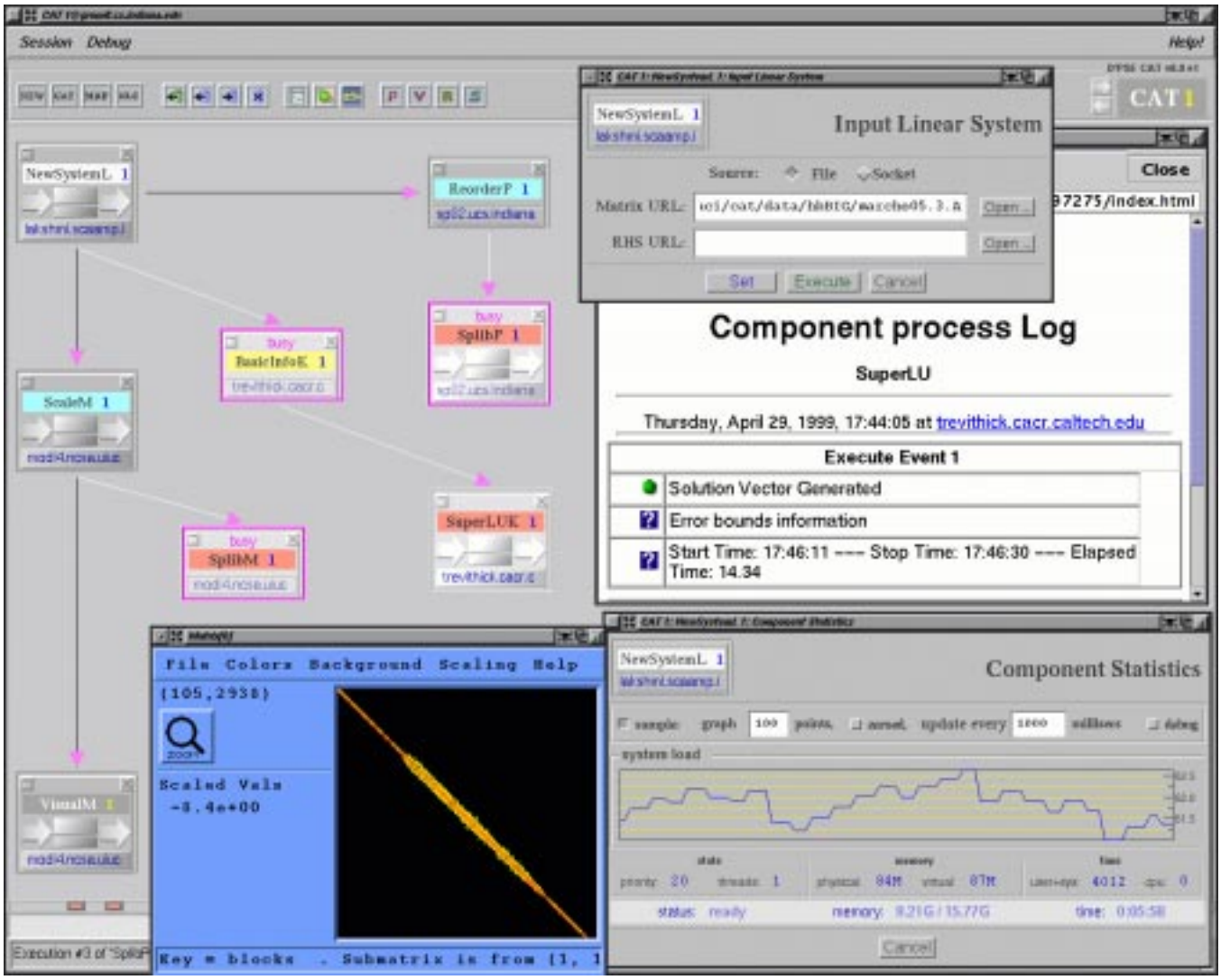

Figure 3. CAT applied to a Linear System Analyzer (LSA) problem

References

[1] A. Thomas, Patricia Seybold Group. Enterprise javabeans technology: Server component model for the java platform, December 1998. http://java.sun.com/products/ejb/whitepaper.html.

[2] Advanced Visual Systems, Inc. AVS: Data Visualization Software, 1998. http://www.avs.com/.

[3] D. Bhatia, V. Burzevski, M. Camuseva, G. Fox, W. Furmanski, and G. Premchandran. Webflow - a visual programming paradigm for web/java-based coarse grain distributed computing, June 1997. http://osprey7.npac.syr.edu:1998/iwt98/products/webflow/.

[4] F. Breg, S. Diwan, J. Villacis, J. Balasubramanian, E. Akman, and D. Gannon. Java RMI Performance and Object Model Interoperability: Experiments with Java/HPC++ Distributed Components. In *Concurrency Parctice and Experience, Special Issue from the Fourth Java for Scientific Computing Workshop*. John Wiley and Sons, Ltd., 1998.

[5] F. Breg and D. Gannon. Compiler Support for an RMI Implementation using NexusJava. Technical Report 500, Computer Science Dept., Indiana University, 1997.

[6] K. M. Chandy, A. Rifkin, P. A. Sivilotti, J. Mandelson, M. Richardson, W. Tanaka, and L. Weisman. A World-Wide Distributed System Using Java and the Internet. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society Press, August 1996.

[7] D. Chappell. *Understanding ActiveX and OLE*. Microsoft Press, 1997.

[8] Common Component Architecture Forum. CCA Component Specification, 1998. http://z.ca.sandia.gov/ cca-forum/gport-spec.

[9] J. Dongarra and H. Casanova. NetSolve, 1998. http://www.cs.utk.edu/netsolve/.

[10] I. Foster and C. Kesselman. Globus Project, 1998. http://www.globus.org/.

[11] I. Foster and C. Kesselman eds. *The GRID Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[12] Khoral Research, Inc. Khoros Software Development, 1998. http://www.khoral.com/khoros.

[13] M. J. Lewis and A. Grimshaw. The Core Legion Object Model. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society Press, August 1996.

[14] NAG. IRIS Explorer, 1998. http://extweb.nag.co.uk/.

[15] OMG et. al. CORBA Components: Joint Revised Submission, December 21, 1998. ftp://ftp.omg.org/pub/docs/orbos/98-12-02.pdf.

[16] M. S. S. Sekiguchi. Ninf: A Network based Information Library for Global World-Wide Computing Infrastructure, 1998. http://ninf.etl.go.jp/.

[17] C. Szyperski. *Component Software: Beyond Object-oriented Programming*. Addison-Wesley, 1998.

[18] University of Utah. SciRun: A Scientific Programming Environment for Computational Steering, 1998. http://www.cs.utah.edu/ sci/scirun/.

[19] M. van Steen, P. Homburg, and A. Tanenbaum. Globe: A Wide-Area Distributed System, January-March 1999.