Technologies for Ubiquitous Supercomputing: A Java Interface to the Nexus Communication System

Ian Foster, George K. Thiruvathukal, and Steven Tuecke
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, U.S.A.
{foster,thiruvat,tuecke}@mcs.anl.gov
http://www.globus.org/

Abstract

We use the term ubiquitous supercomputing *to refer to systems that integrate low- and mid-range computing systems, advanced networks, and remote high-end computers with the goal of enhancing the computational power accessible from local environments. Such systems promise to enable new applications in areas as diverse as smart instruments and collaborative environments. However, they also demand tools for transporting code between computers and for establishing flexible, dynamic communication structures. In this article, we propose that these requirements be satisfied by introducing Java classes that implement the global pointer and remote service request mechanisms defined by a communication library called Nexus. Java supports transportable code; Nexus provides communication support and represents the core communication framework for Globus, a project building infrastructure for ubiquitous supercomputing. We explain how this NexusJava library is implemented and illustrate its use with examples.*

1 Introduction

Rapid advances in networking technologies have made it possible to construct an application that integrates resources located at multiple geographically distributed locations. Various high-end networking experiments have demonstrated convincingly that important new classes of applications become possible in such environments [3]. Typically, these applications exploit high-speed networks to assemble in one (virtual) place collections of resources that would not otherwise be accessible, such as scientific instruments, supercomputers, databases, and people.

Most work on high-performance distributed computing has originated within the high-performance computing community, and these origins are reflected in the types of applications considered and the techniques used to construct these applications. Supercomputers are highly visible, and programs typically use message passing to transfer data between program components. The user interfaces with the application from a local system—or, in many cases, from a high-end display device [3]. While effective, these techniques have the drawback that they hinder the widespread dissemination of the technology, for example because sophisticated software systems must be installed at each participating site [7].

An alternative model for high-performance distributed computing focuses on making the power of remote supercomputers accessible to users in a completely transparent manner. The goal is to support the development of applications that execute locally (whether on a low-end PC or high-end workstation) and exploit remote supercomputing resources to provide enhanced services. We use the term *ubiquitous supercomputing* to denote this type of computing, because by coupling low-cost local devices with remote supercomputer resources, it combines aspects of ubiquitous computing [15] and traditional supercomputing.

This article is concerned with the tools that might be used to construct ubiquitous supercomputing systems and applications. We explain how a combination of the Java programming language and two simple mechanisms—the global pointer and remote service request—can be used to satisfy these requirements.

2 Ubiquitous Supercomputing

We discuss the types of applications that might be constructed in a ubiquitous supercomputing system.

Smart instruments. The utility of many scientific instruments can be enhanced significantly by the use of computational techniques. For example, in the case of an imaging device, computers can be used to enhance images, to annotate images with hints as to significant features, to locate similar images, to provide comparisons of observation and theory, or to integrate information from several imaging modalities. Such techniques have been used to a limited extent for some time; however, in general, only fairly limited computation could be performed because it was not feasible to co-locate a high-end computer with the instrument. The advent of high-speed networks makes it feasible to use a single supercomputer to serve many instruments, with the result that the computational power accessible to a single instrument increases dramatically. Quasi-real-time computer-enhanced imaging becomes possible.

Lee et al. [13] have developed an interesting example of this type of application. The instrument in question, a weather satellite, takes pictures at multiple wavelengths. Data from the satellite is received at the ground station and passed over a wide area network to a supercomputer, where it is enhanced by a cloud detection algorithm to obtain three-dimensional images of cloud location. These images are then passed to a display device, allowing scientists to browse the computer-enhanced images almost in real time.

Smart applications. Similar techniques can be used to enhance the utility of desktop applications. Currently, these may have sophisticated user interfaces but perform relatively simple computations. The ability to connect to substantially greater computing resources can allow desktop applications to perform more demanding computations. For example, a future spreadsheet might connect to a model of the U.S. economy when evaluating investment strategies, or to a climate model when evaluating risk management strategies for an agricultural concern. A system for preparing audio-visual presentations might reach over the network to search massive image banks for pictures matching a specified textual description or might exploit external computing resources to render a video clip.

Simple examples of this sort of tool have already been constructed. To name just two examples, the Network Enabled Optimization System (NEOS) allows users to submit optimization problems electronically to an optimization server, while NetSolve allows desktop applications written in MatLab to pass computationally demanding tasks to high-performance computers. In both these cases, access to networked resources is far from seamless; however, these systems are suggestive of how future "smart applications" might work.

Collaborative environments. Collaborative environments are computer systems that enhance people's ability to collaborate with people at remote locations. A wide variety of such systems exist, ranging from systems focused on enhancing people's ability to create shared documents (e.g., Lotus Notes) to those designed to permit more free-form electronic discussions in shared virtual spaces (e.g., MUDs). Advanced collaborative environments enable users to collaborate in the manipulation of complex virtual spaces, which may furthermore incorporate entities corresponding to supercomputer simulations. For example, the BoilerMaker system [4] allows engineers at multiple locations to participate in the placement of injection devices in a simulated combustion system. The complete system comprises multiple display devices and a supercomputer, connected by high-speed networks.

Looking further into the future, Gelertner posits the widespread deployment of what he calls Mirror Worlds [11], computer models of interesting aspects of reality designed to make those aspects of reality more readily visible to people—and perhaps also to simplify management. (Examples might include a city government, hospital, or traffic system.) These systems would include advanced computer models, data assimilation from many sensors, and collaborative capabilities allowing explorers of a mirror world to communicate with each other.

3 Ubiquitous Computing Technologies

To a significant extent, the hard technical problems underlying the applications described in the preceding section are those of distributed computing. However, two aspects of these applications complicate the picture. First, to a much greater extent than in most distributed applications, these applications are performance focused. For example, a supercomputer-enhanced microscope that is intended to provide real-time response needs to be able to acquire computational resources rapidly when an image is available, and then transfer large amounts of data to that resource for processing. The second difference is that true ubiquity demands tools that can be deployed quasi-universally. Many of the example applications referred to above require that sophisticated software be installed locally before a user can exploit remote computing capabilities. This requirement severely limits our ability to disseminate the technology.

The Web provides a compelling case study for how to achieve universal access to a highly distributed service. The beauty of the Web is that anyone with a browser can use it to access information anywhere in the world. The key to this universal access is the provision of a low-cost, standard interface mechanism (the browser) that is dynamically extensible (we just upload an HTML document) to reflect the characteristics of a remote data source.

While tremendously flexible as a tool for locating and browsing multimedia data, the original Web protocols were constrained by the fact that the browser could not perform computation: it could only fetch and display data. The Java programming language [1] represents one step toward overcoming this limitation. Java is a simple object-oriented programming language (with similarities to Objective C and C++), augmented with standard libraries for graphics, communications, and other functions. Java programs can be compiled to byte codes to obtain a portable, reasonably compact representation suitable for communication over networks. A process receiving Java byte codes can execute them by using an interpreter or just-in-time compiler. Java interpreters have been embedded in various Web browsers, making it possible for users to create Web pages that perform various computations.

While Java has significant advantages as a language for ubiquitous computing, it is deficient in the important area of communication. (Other significant shortcomings, for example, in security area are beyond the scope of this article, which focuses on communication frameworks.) The Java library provides only basic support for communication using low-level UDP and TCP protocols. The lack of higher-level communication mechanisms greatly complicates the implementation of applications such as those described above.

We argue that communication facilities for Java should satisfy four basic requirements. (1) *Asynchrony.* While synchronous remote procedure call (RPC) is appropriate for many distributed applications, particularly those with a client-server structure, high-performance ubiquitous supercomputing applications also require mechanisms that do not enforce synchronization between sender and receiver, such as asynchronous remote function invocation and—in some cases—point-to-point communication (message passing). (2) *Symmetry.* "Clients" (user Java programs) and "servers" (remote processes) need to be able to be equal partners in a computation. Not only should a client be able to call procedures in a server, but vice versa also. (3) *Global names.* The ability to create references to objects and then communicate those references between objects proves to be extremely useful in practice, making it possible to create complex, distributed data structures and to write programs that operate on these data structures in a uniform fashion, independently of object location. Note that what is required here is a global name space, not a global address space. (4) *High performance.* We require techniques that permit high-performance implementations. This requirement means not only that our techniques should not introduce performance bottlenecks, but they should permit us to write programs that can adapt their behavior to the often complex heterogeneous systems in which they can be expected to operate.

As we explain in the next section, we propose to meet these requirements by developing a Java binding for a communication library called Nexus that provides remote object reference (called, in Nexus, a global pointer) and asynchronous remote method invocation (in Nexus, remote service request) mechanisms.

4 Nexus

Nexus is a communication library developed at Argonne National Laboratory and the California Institute of Technology to support applications that require mechanisms for asynchronous communication, multithreading, and dynamic resource management in heterogeneous environments [10].

Nexus services provide direct support for lightweight threading, address space management, communication, and synchronization. The Nexus interface is structured in terms of five basic abstractions, illustrated in Figure 1: nodes, contexts, threads, global pointers, and remote service requests. A computation executes on a set of *nodes* and consists of a set of *threads*, each executing in an address space called a *context*. For the purpose of this article, it suffices to assume that a context is equivalent to a process and that a node is equivalent to a particular computer.

A *global pointer* (GP) is a name that can refer to a memory location (or object) located anywhere in a distributed system. GPs are used in conjunction with asynchronous *remote service requests* (RSRs) to invoke actions at remote locations. An RSR takes a GP, a procedure name, and data; transfers the data to the context referenced by the GP; and remotely invokes the specified procedure, providing the data and the local portion of the GP as arguments. GPs can be passed as arguments to RSRs, hence allowing global names to be propagated between processes.

Experience indicates that Nexus mechanisms can be implemented efficiently on a wide range of parallel and networked computer systems [10]. Furthermore, global pointers can be used as a basis for mechanisms that support both automatic and programmer-guided selection from among multiple communication methods [6]. These mechanisms allow programs to execute efficiently in heterogeneous environments and make it possible to use different communication protocols for different communication structures. Nexus has been used to implement a variety of different parallel and distributed programming tools providing different interaction models, including remote procedure call (in CC++ [2] and nPerl, an RPC library for the Perl scripting language), multimedia streams (in CAVEcomm [5]) and message passing (the Message Passing Interface [8]). Nexus also serves as the communication infrastructure for the Globus distributed computing infrastructure toolkit [9].

Nexus mechanisms satisfy each of the requirements introduced above. The RSR provides an asynchronous communication substrate, on which can be layered a variety of more sophisticated interaction methods. The global pointer makes it easy to specify symmetric structures, since a "client" can easily pass a global pointer to a "server," hence allowing the server to invoke procedures in the client. Global pointers also provide a global name space. Finally, Nexus mechanisms have been shown to permit high-performance implementations.

5 A Java Binding for Nexus

We have constructed a Java binding for Nexus; that is, an interface to Nexus mechanisms that allows Java programs to create and exchange global pointers and to perform remote service requests to methods defined in objects referenced by these global pointers. This binding also allows Java programs to communicate with other programs (such as MPI or one of the many parallel languages that support Nexus) that employ Nexus mechanisms.

The Java binding for Nexus implements just the Nexus global pointer and remote service request mechanisms. Nexus also includes support for a set of thread management, condition variables, and mutual exclusion (mutex) functions; however, these functions need not be implemented in the Java binding for Nexus, because the Java Thread class supports these functions and the Java language itself provides support synchronization mechanisms at the object and method levels.

As we shall explain, the Java binding provides direct access to the relatively low-level Nexus interface; this interface can then be used to build higher-level Java communication libraries for specific purposes.

We implement the Java binding as a Nexus-compatible library written entirely in Java. This means that Nexus code can run within any system that incorporates a Java interpreter or just-in-time compiler. The library comprises four basic classes: `Nexus`, which supports initialization, argument handling, handler registration, global pointer creation, and attachment to other processes; `GlobalPointer`, which implements the Nexus global pointer abstraction, for use in remote service requests; `PutBuffer`, which provides mechanisms for buffer packing; and `GetBuffer`, which provides buffer unpacking mechanisms. We shall use a simple example to illustrate the use of the various functions defined in these classes. (NexusJava function prototypes are generally equivalent to those of the Nexus C library.)

Our example comprises the simple client and server programs in Figure 2. The client performs a single remote service request to the server. The client terminates immediately after generating the request, and the server terminates immediately after performing the request. This trivial example does not really demonstrate the expressiveness of NexusJava, but does have the pedagogical advantage of introducing most NexusJava features.

The client begins by instantiating and initializing a `Nexus` object. This must be done before any other NexusJava operations are performed. The client then attaches to the server using the `Nexus.attach()` method. This method takes as its argument a URL specifying the hostname and port on which the server is listening; it returns a `GlobalPointer` referencing an object in the server process.

Once the client has attached to the server, it can use the GP to invoke methods defined in the remote object that this pointer references. For example, the procedure `call_server_handler()` invokes a remote procedure called `server_handler`, passing as its argument the single integer 10. It calls low-level Nexus routines to (a) initiate the remote service request, (b) construct a buffer containing the integer argument, and (c) complete the RSR. The client then uses the `GlobalPointer.destroy()` method to destroy the GP to the server; this action severs the connection between the client and server. Finally, the client shuts down NexusJava by calling the `destroy_current_context()` method on the `Nexus` object. This action cleanly terminates any threads and other state that are maintained by this object.

The server program, like the client, first instanti-

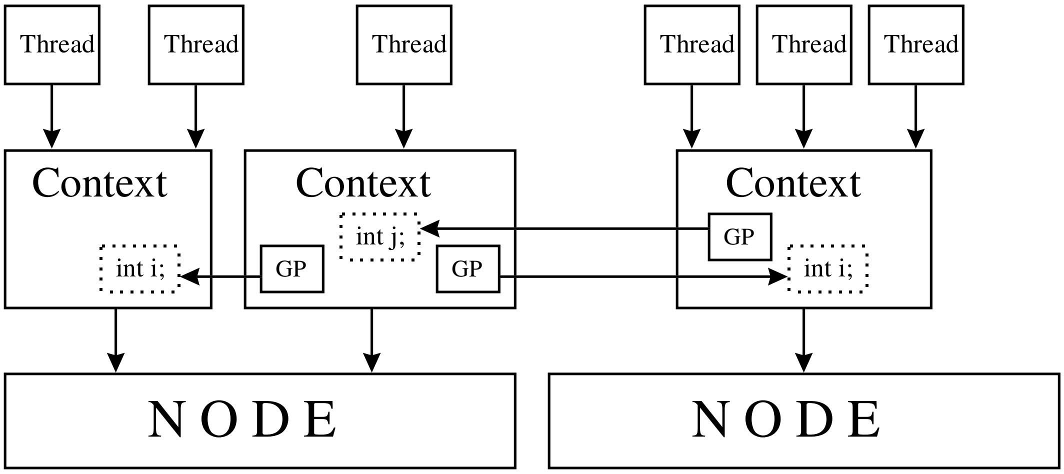

Figure 1. Nodes, Contexts, Threads, and Global Pointers

```
public class ExampleClient {
    private Nexus nexus;
    public static void main (String args[]) {
        ExampleClient n = new ExampleClient(); n.start(args);
    }
    public void start(String args[]) {
        GlobalPointer gp;
        nexus = new Nexus();
        args = nexus.init(args, "nx", null);
        try { gp = nexus.attach("x-nexus://cosmo.mcs.anl.gov:1234/");
              call_server_handler(gp, 10);
              gp.destroy();
        } catch (Exception e) e.printStackTrace();
        nexus.destroy_current_context(false);
    }
    public void call_server_handler(GlobalPointer gp, int i) {
        PutBuffer buffer;
        try { buffer = gp.init_remote_service_request("server_handler", 42);
              buffer.set_buffer_size(buffer.sizeof_int(1), 1);
              buffer.put_int(i);
              buffer.send_remote_service_request();
        } catch (Exception e) e.printStackTrace();
    }
}
```

Figure 2. Example: Client program that demonstrates initialization, packing a buffer, and sending an RSR.

```
public class ExampleServer implements HandlerInterface,
AttachApprovalInterface {
    private Nexus nexus;
    private GlobalPointer this_gp;
    public static void main (String args[]) {
        ExampleServer n = new ExampleServer(); n.start(args);
    }
    public void start(String args[]) {
        nexus = new Nexus();
        args = nexus.init(args, "nx", null);
        register_my_handlers();
        this_gp = nexus.global_pointer(this);
        nexus.allow_attach(1234, this);
        wait_for_client(); nexus.disallow_attach(1234);
        this_gp.destroy(); nexus.destroy_current_context(false);
    }
    public void register_my_handlers() {
         Handler h[] = new Handler[2];
         h[0] = new Handler("server_handler",42,Handler.NEXUS_HANDLER_TYPE_THREADED,this,0);
         h[1] = new Handler("other_handler",53,Handler.NEXUS_HANDLER_TYPE_NONTHREADED,this,1);
         nexus.register_handlers(h);
    }
    public void invoke_handler(String name,int id,int local_id,Object addr,GetBuffer buf) {
        switch (local_id) {
           case 0:
              try { int i = buf.get_int();
                    server_handler(i);
              } catch (Exception e) e.printStackTrace();
              break;
           case 1:
              other_handler();
              break;
        }
    }
    public GlobalPointer attach_approval(String url) {
        return(this_gp);
    }
    private synchronized void wait_for_client() {
        try { wait(); } catch (Exception e) e.printStackTrace();
    }
    private synchronized void server_handler(int i) {
        System.out.println("server_handler() got i="+i);
        try { notify(); } catch (Exception e) e.printStackTrace();
    }
    private void other_handler() {}
}
```

Figure 3. Example: Server program that demonstrates handler registration, handler invocation, and buffer unpacking.

ates and initializes a `Nexus` object. Then, it registers the set of handler names for which it will accept messages. The registration is performed by the routine `register_my_handlers()`, which creates an array of `Handler` objects in which each element describes a handler. This description includes the handler name (e.g., "server_handler"), a handler id (e.g., 42), a flag specifying whether this handler should be invoked in a newly created thread or in an existing thread, the `HandlerInterface` object to call when an RSR arrives for this handler, and a local handler id that can be used for quick dispatch of the handler within that `HandlerInterface` object. The `Nexus.register_handlers()` method is then called with the `Handler` array to inform the `Nexus` object of the handlers for which RSRs are to be accepted.

After registering the handlers, the server next calls `Nexus.allow_attach()` to indicate that it is prepared to accept incoming RSRs. It then suspends in `wait_for_client`, processing subsequent attachment or RSR requests as call backs. Attachment requests result in calls to the `attach_approval()` method in the `AttachApprovalInterface` object passed as the second argument to `allow_attach()`. The `attach_approval()` method returns a GP to a local object, which will be returned to the attacher. The server may also decide to deny the attachment request, in which case it must return `null`.

RSR requests (for example, to `server_handler`) cause the `invoke_handler()` method (part of the `HandlerInterface` provided by `ExampleServer`) to be called by NexusJava. This method (a) uses the handler name, id, and local id to figure out which of this object's methods should be invoked, (b) unpacks the `GetBuffer` to get the arguments for the method, and (c) calls that method with the arguments.

As mentioned above, handlers can be either threaded or nonthreaded. When an RSR arrives for a threaded handler, a new Java thread is created by NexusJava, and the `invoke_handler()` method is called from within this new thread. There are no restrictions on what this handler may do. NexusJava also supports a more efficient but restricted form of handler invocation. If a handler is registered as nonthreaded, NexusJava does not create a new thread. Instead, it calls `invoke_handler()` directly from its preexisting, internal communications thread. This approach avoids the cost of thread creation and switching during handler dispatch. However, the user must guarantee that a handler registered as nonthreaded will not block (wait) on any operation that may require another RSR handler invocation to unblock (notify) the first handler.

Once the server receives the RSR and calls the `server_handler()` method, this method will notify the main thread waiting in `wait_for_client()`. The server then disallows additional attachments by calling `Nexus.disallow_attach()` and shuts down NexusJava using `Nexus.destroy_current_context()`.

In summary, the NexusJava library makes the full power of Nexus available to Java programs, which can use Nexus mechanisms to create global pointers to objects, pass these references between processes, use RSRs to invoke methods defined in remote objects, and so forth.

6 Higher-Level Interfaces

As noted above, a wide variety of higher-level interaction models can be layered on top of the low-level Nexus mechanisms. Here, we discuss techniques that can be used to implement an RPC model. The basic idea is to use IDL-like techniques to generate automatically the code responsible for registering handlers, marshaling arguments to remote method calls, demarshaling arguments, and dispatching method invocations. Similar techniques are used in other systems, notably CC++ [2] and CORBA [12].

Figures 2 and 3 illustrate what is involved. In the `ExampleClient` class, the `call_server_handler()` method is essentially a stub that encapsulates the argument marshaling and other bookkeeping required to perform a remote method invocation to the `server_handler()` in the `ExampleServer`. Similarly, in the `ExampleServer` class, the `invoke_handler()` method is essentially a stub that demarshals the arguments from the buffer and calls the appropriate method (such as `server_handler()`) locally.

These stub methods can be generated automatically in a number of different ways. The CORBA approach could be followed, whereby a high-level Interface Definition Language (IDL) is used to describe the methods to which one wishes to perform remote invocations. An IDL compiler is then used to convert automatically this IDL specification into Java stub code. A disadvantage of this approach is that the definition and compilation of explicit interfaces can be rather complex. Since the Java source to byte-code compiler is implemented in Java, and since Java classes can be loaded on the fly, an intriguing alternative is to generate the appropriate stubs on the fly when doing handler registration.

7 Other Approaches

The Java community has seen several recent attempts to provide higher-level communication in Java.

The two most important (and interesting) are CORBA-based products by several companies and JavaSoft's Remote Method Invocation (RMI) package [14].

The Common Object Request Broker Architecture (CORBA) provides standard mechanisms for exporting objects for remote use, for locating remote objects, and for invoking methods in remote objects. As mentioned above, objects export interfaces defined using an IDL, which is compiled into language specific stubs for use in remote method invocation. IDL to Java mappings have been defined, and several companies have released an IDL compiler. These products allow Java objects to communicate with other remote objects that have been written in Java or another language.

The JavaSoft Remote Method Invocation (RMI) specification is similar in spirit to the CORBA approach, with three significant differences. First, it uses Java-specific interface definitions instead of a language-neutral IDL specification to produce stub code. This is a sensible design decision for an all-Java application focus, but hinders interoperability. Second, RMI does not use standard CORBA methods for object location and method invocation. However, once a reference to a remote object has been obtained, both Java implementations of CORBA and RMI allow methods to be invoked on that object using essentially the same syntax as normal, local Java method invocations. Third, the RMI specification defines a Java-specific framework for marshaling parameters between locations. This Object Serialization framework is tightly coupled with the compiler front-end. Like RMI, it works well when the entire application is to be written in Java, but is not easily integrated with other languages, such as C and C++.

CORBA and RMI mechanisms can be used to provide Nexus-like functionality, namely, the abilities to obtain references to remote objects and to use those references to invoke methods within those objects. The JavaSoft CORBA and RMI products are better integrated into Java than NexusJava. However, they also have significant limitations. Neither CORBA nor RMI supports the fully asynchronous operations provided in Nexus. CORBA does not support the concept of a global pointer and hence cannot define a global name space. RMI supports a remote object construct that has some similarities to the global pointer, but it is Java-specific and does not support interfaces to other systems.

8 Conclusions

We have shown how the Nexus global pointer and remote service request mechanisms can be incorporated into Java by defining appropriate Java classes. The resulting system makes it possible to construct the extremely flexible communication structures enabled by Nexus, without compromising the transportability of Java code. The techniques also support interoperability with other Nexus-based applications. Our next steps in this area will be to experiment with the use of NexusJava for a range of ubiquitous supercomputing applications. We are also interested in developing higher-level interfaces to Nexus mechanisms by using some of the techniques introduced above.

Our work on Nexus forms part of a larger project called Globus [9] that is developing key infrastructure components for high-performance distributed computing. We expect availability of NexusJava to increase significantly the range of applications for which Globus services are useful.

For more information on the NexusJava project and the current software distribution, see the Nexus home page `http://www.mcs.anl.gov/nexus/`. More information on the Globus project can be found at the Globus home page `http://www.globus.org/`.

Acknowledgments

The Nexus library used to construct NexusJava has been developed jointly with Carl Kesselman. This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

Thanks to Gail Pieper and Gregor von Laszewski for their assistance in preparing the final manuscript.

References

[1] K. Arnold and J. Gosling. *The Java Language Specification*. Addison-Wesley, 1996.

[2] K. M. Chandy and C. Kesselman. CC++: A declarative concurrent object oriented programming notation. In *Research Directions in Object Oriented Programming*, pages 281–313. The MIT Press, 1993.

[3] T. DeFanti, I. Foster, M. Papka, R. Stevens, and T. Kuhfuss. Overview of the I-WAY: Wide area visual supercomputing. *International Journal of Supercomputer Applications*, 10(2):123–130, 1996.

[4] Darin Diachin, Lori Freitag, Daniel Heath, James Herzog, William Michels, and Paul Plassmann.

Remote engineering tools for the design of pollution control systems for commercial boilers. *International Journal of Supercomputer Applications*, 10(2), 1996.

[5] T. L. Disz, M. E. Papka, M. Pellegrino, and R. Stevens. Sharing visualization experiences among remote virtual environments. In *International Workshop on High Performance Computing for Computer Graphics and Visualization*, pages 217–237. Springer-Verlag, 1995.

[6] I. Foster, J. Geisler, C. Kesselman, and S. Tuecke. Multimethod communication for high-performance metacomputing applications. In *Proceedings of Supercomputing '96*. ACM Press, 1996.

[7] I. Foster, J. Geisler, W. Nickless, W. Smith, and S. Tuecke. Software infrastructure for the I-WAY high-performance distributed computing experiment. In *Proc. 5th IEEE Symp. on High Performance Distributed Computing*, pages 562–571. IEEE Computer Society Press, 1996.

[8] I. Foster, J. Geisler, and S. Tuecke. MPI on the I-WAY: A wide-area, multimethod implementation of the Message Passing Interface. In *Proceedings of the 1996 MPI Developers Conference*, pages 10–17. IEEE Computer Society Press, 1996.

[9] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 1997. to appear.

[10] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37:70–82, 1996.

[11] David Gelertner. *Mirror Worlds*. Oxford University Press, 1991.

[12] Object Management Group. Common object request broker architecture specification. http://www.omg.org.

[13] C. Lee, C. Kesselman, and S. Schwab. Near-realtime satellite image processing: Metacomputing in CC++. *Computer Graphics and Applications*, 16(4):79–84, 1996.

[14] Sun Microsystems. Remote method invocation and object serialization specifications. http://www.javasoft.com.

[15] Mark Weiser. Hot topics: Ubiquitous computing. *IEEE Computer*, 26(10), October 1993.