

Design of Distributed Component Frameworks for Computational Grids

Madhusudhan Govindaraju, Himanshu Bari, Michael J. Lewis
Department of Computer Science, SUNY Binghamton, NY, 13902
{mgovinda, himanshu, mlewis}@cs.binghamton.edu

Abstract

The Common Component Architecture (CCA) defines a specification for the implementation of frameworks to support component-based high performance applications. The same framework specification is intended to describe different implementations for different environments, ranging from sequential processes to parallel processors to wide area distributed systems or Grids. The mapping of the CCA specification to an underlying runtime environment presents important design challenges. We discuss these design issues for distributed Grid frameworks. We identify key design constraints and use them to define the design space. We then describe two different concrete implementations of the CCA specification, CCA Legion and XCAT.¹

Key Words: Common Component Architecture (CCA), Grid Computing, Legion, XCAT

1 Introduction

Computational Grids [4] consist of hardware and software infrastructure that includes a set of network accessible services and resources that provide a seamless computing environment. Grids will simplify the development and deployment of large scale scientific applications. The realization of this vision depends on the design and development of a distributed framework that can present the environment in a dependable, consistent, and pervasive manner. Such a framework must shield users from the details of managing the scale and complexity of Grid-based scientific applications. Distributed frameworks can be designed in various ways to meet these requirements. In this paper, we discuss the design of two different distributed frameworks that use the component design paradigm. Both frameworks adhere to the same high level specification, thereby facilitating their eventual interoperability, but use different mechanisms and approaches to realize their functionality.

¹This research is supported by DOE Grant DE-FG02-02ER25526 and NSF Career Award ACI-0133838.

The use of component-based design for developing complex HPC applications has found favor with both academia and industry. The business world has developed systems that include Microsoft COM, Distributed COM (DCOM), .NET, Corba Component Model (CCM), and Enterprise Java Beans. Component based systems foster code re-use and provide high level abstractions to shield users from low level details. Applications can be developed using plug-and-play assembly of components; scientists can manipulate a palette of components to dynamically change the solution strategy by swapping components into and out of their applications at run time.

A consortium of university and national laboratory researchers have formed the “CCA Forum” [2], which has designed a component specification for large scale scientific computation. The CCA Forum has developed the Common Component Architecture (CCA) specification, a simple yet powerful specification to support high performance components for both sequential and distributed applications. The CCA specification defines the roles and functionality of entities necessary for high performance component-based application development, including components, ports, and services for instantiating and connecting them. Different implementations of the specification provide different frameworks for a variety of contexts, including sequential, parallel, and distributed environments. Section 2 elaborates on the important design aspects of CCA, and the remainder of this paper discusses the implementation of the CCA specification for distributed frameworks that support Grid-based applications.

Thus, the contribution we make in this paper is to provide the interpretation of an important emerging standard for component-based high performance scientific computation, in a distributed Grid environment. The CCA approach is gaining favor within DOE-based (and other) national laboratories, and in the scientific computing community in general [7]. CCA is of particular importance to the Department of Energy (DOE)’s Center for Component Technology for Terascal Simulation Software (CCTTSS), which is funded as an Integrated Software Infrastructure Center (ISIC) under the Scientific Discovery through Advanced Computing (SciDAC) program. The DOE Office of Science recognized CCA

as one of its top 10 science achievements of 2002 [9].

By discussing the design of our two separate CCA Grid frameworks, we demonstrate that the high level CCA specification, which has been successfully mapped to parallel and sequential frameworks [1], can also be used for distributed Grid-based environments. As a result, scientific programmers can more easily make use of Grid resources, because they can target a CCA-based framework that is familiar and easy to use.

There are two kinds of approaches to using the CCA design to develop distributed Grid frameworks: (1) develop a new distributed Grid system that is based entirely on the CCA specification, or (2) take an existing distributed system and add CCA functionality to make it compliant with the specification. We explain these approaches in detail with our work on XCAT [5] and Legion [8].

We begin with an overview of the CCA specification and its key concepts in Section 2. Section 3 then discusses requirements for implementing the CCA specification in a distributed environment; we focus on those aspects of the CCA specification that require implementations that are significantly different from those for parallel or sequential CCA frameworks. Section 4 identifies a set of questions that help define the design space for distributed framework implementation. The CCA concepts, framework requirements, and design space definition combine to frame the discussion of two distinct CCA-compliant distributed framework designs, which we present in Section 5. Section 6 summarizes the paper and discusses directions for future work.

2 Common Component Architecture

The Common Component Architecture (CCA) specification is designed to provide a high-performance plug-and-play environment for large scale scientific applications. It facilitates the import of existing scientific code into a CCA compliant framework by imposing only a small number of requirements on component developers. The CCA model consists of two fundamental entities: *components* and *frameworks*. Components encapsulate pieces of code that implement a certain functionality, and are the units of software that can be composed to build applications. The framework provides the required milieu in which components run and interact with one another.

The component specification of CCA is expressed as a set of interfaces that define the required behavior for component-to-component and component-to-framework interaction. However, the specification does not mandate nor suggest how the framework itself is constructed. This approach explicitly allows research groups to experiment with different ideas and to develop specialized frameworks. For example, it is possible to develop frameworks that are tai-

lored for wide-area networks and also those that are highly specialized for single-process interactions. The most important CCA concepts are: (1) *ports*, to define interfaces through which components communicate with one another; (2) *services objects*, to manage a component's ports; (3) *component identifiers*, to serve as opaque handles for components in the framework, and (4) *builder service*, to create, destroy, and connect an application's components. Each of these concepts is described separately below; together, their implementation in some environment comprises a CCA framework.

Ports: CCA applications are composed by connecting components to one another via their *ports*, which represent both component interfaces and interface requirements. That is, there are two kinds of ports: *provides* ports and *uses* ports. *Provides* ports are sets of functions that represent the services exported by the component to other components, and *uses* ports represent the services that the component needs to call. The mechanism by which components call one another through ports is left unspecified by the CCA specification, because the component interaction mechanism depends on the framework. Port descriptions for CCA components are provided using the Scientific Interface Description Language (SIDL) [7], which is specifically designed for high performance scientific applications. SIDL explicitly supports complex numbers, dynamic multi-dimensional arrays, parallel attributes, and communication directives. An expected CCA usage paradigm has component developers specifying their component interfaces and characteristics in SIDL, using code generation tools to generate framework-specific wrappers for their component code, and then filling in implementations within those wrappers. Thus, the same SIDL description for a component would be valid across all CCA-compliant framework implementations.

Services Object: Each component contains a *Services Object* that manages the component's ports. CCA allows ports to be dynamically added and deleted through its Services Object. The Services Object also satisfies its component's requests for handles to specific ports in other components. The Services Object encapsulates the details of the framework specific bindings for its *provides* and *uses* ports. The CCA specification provides APIs for the Services Object that include functions to register and retrieve ports by name and type.

Component Identifier: The ComponentID serves as an opaque handle to a component. CCA does not mandate a format for this handle, thereby allowing each framework to design the handle in a format that is best suited for its needs. In the single process case, it is expected that the handle will be a local pointer, for efficiency. However, in distributed frameworks the ComponentID serves as a global pointer to the component. The ComponentID plays a central role in component assembly. In most frameworks the ComponentID points to the Services Object of the component, so that

it can be used effectively for connection and introspection of components.

Builder Service: The Builder Service is charged with implementing the following types of operations:

Create: The Builder Service presents a uniform API for all components to instantiate other components. Once a component has been instantiated, the service returns a ComponentID to the new component. This ComponentID can then be used to directly communicate with the component.

Destroy: The Builder Service provides an operation to destroy component instances. The component's internal functional logic dictates the course of action to be taken when a destroy request has been received. The Create and Destroy operations on the Builder service can be compared to the "factory" design pattern.

Connection: The Builder Service allows programmers to connect the ports of components, thus facilitating component assembly. Consequently, this service can also be used to develop builder tools that include Graphical User Interfaces (GUIs) and scripting front-ends. Component-connections take place between *uses* ports and *provides* ports. Two ports are said to be compatible (and can therefore be connected) if they have same the SIDL type [7].

Introspection: Introspection of CCA components can be done by querying the component's Builder Service. This introspection can retrieve the list of ComponentIDs for the components in that process along with the ports and their properties. This allows programmers to make decisions at runtime about which components to include in their applications, and enables tools that check for port type compatibility while components are being assembled.

3 Important Characteristics of Distributed Frameworks

By definition, a distributed CCA-compliant application spans multiple address spaces; components of a distributed CCA application are not necessarily co-located within the same process, as they are in sequential and parallel frameworks. This property influences the design of distributed frameworks, because it implies remote invocation and remote creation. That is, inter-component function calls must be able to cross address space and machine boundaries, and applications developers must be able to use remote process creation and application management facilities to instantiate new components on remote machines, and to coordinate applications that span multiple machines. Furthermore, the fact that components run within a single federated distributed system presents the opportunity to improve *registration* and *discovery*. Components should be able to register themselves

within the framework, and be easily discovered by other entities running anywhere within the framework.

In some sense, each of these serves as a constraint on the design space of distributed component frameworks. We elaborate on how *remote creation* and *remote connection* constrain the design space by discussing them further, and identifying the features that are needed to support them.

Remote Invocation: Each distributed system imposes a set of requirements on its remote communication substrate. These requirements are based on the set of objectives and the wide spectrum of applications that the distributed system is designed to support. Examples of communication facilities include Message Passing, RPC and RMI. In each case, CCA requires a specialized representation for "handles" to the ports and services of components. In the single process case it suffices to use a local pointer as the handle. However, in the distributed case, these handles need to be designed as "global pointers" that point to remote address spaces. Often these handles are designed as global references: objects that function as proxies for remote objects.

The requirement that CCA ports and services be invoked remotely across address spaces and machine boundaries, then, implies the following necessary features within the framework: (1) *global namespace* within which to give names to components and services; (2) a mechanism for implementing function calls on ports as *remote function invocations* on components running in different processes on different machines; and (3) a mechanism for *introspection* through which the ports and services available within the framework can be discovered and accessed

Remote Creation: Building distributed applications by component assembly sometimes requires the use of both running components and newly instantiated components. Launching components in remote address spaces involves the establishment of the required environment for the component, including the host name, the location of the executable, the instantiation mechanism, and values for stdin and stdout, for example. Remote creation, then, requires the following features: (1) *Heterogeneity management:* The system may contain hosts with different architectures and operating systems, so the framework must be able to deploy the right component specification in the right environment; (2) *Resource managers* that serve as representatives for their computing resources must be able to service requests and carry them out on local resources; and (3) *Component binaries* must be stored, managed, replicated, and/or moved to support instantiation on a remote platform that may not share the same file system.

4 Key Design Choices for Mapping CCA to Distributed Frameworks

Mapping the CCA specification to distributed frameworks presents a set of choices. In this section, we enumerate these choices as a list of questions and describe the different options that are available for each. In this way, we define the design space for CCA-compliant distributed component frameworks.

How is the CCA ComponentID type represented in a distributed environment? The ComponentID serves as a handle to a component and can be passed to components that live in remote address spaces. Therefore, it must be possible to serialize and deserialize ComponentIDs. This also requires that the ComponentID belong to a namespace that is valid within the entire distributed framework. The ComponentID is primarily used to connect components. In the network of connected components, sometimes there is a need to co-locate two components within the same process, perhaps for security or efficiency reasons. Thus, the ComponentID type should be designed so that it can implement the optimized (co-located) case using local pointers, while allowing serialization and deserialization when necessary for remote interactions.

How is the CCA Port type represented in a distributed environment?: *Uses* and *provides* ports have different roles within a component. *Provides* ports present a service that other components can access, requiring that they be designed with capabilities to serve as a remote service. *Uses* ports are internal to each component and inaccessible to other components. The CCA specification requires that component developers provide implementations of *provides* ports but not necessarily *uses* ports. Users just specify the type of the *uses* port they intend to use. The distributed framework needs to have a factory that can generate internal *uses* ports based on their types.

Where can the key CCA functions be called from? (Are they remote calls, or local?): As mentioned, the CCA specification is designed for use with both distributed and sequential applications. In the sequential case, all functions are implemented as an API available within the address space through local function calls. In a distributed environment, however, some of the calls need to be made locally, some remotely, and some require a combination of the two. Thus, the framework designer must determine how the standard functions in the CCA specification are exposed to programmers, and where the functionality is implemented. What this implies, then, is the fact that some of the methods need remote semantics for distributed frameworks. For example, when a *provides* port is retrieved via the `getPort(string portName)` call on the *Services* object, a serializable port reference needs to be returned. However, for a *uses* port just

the local pointer should be returned. The back-end implementation of these ports should be hidden from the user. For each CCA call, the distributed framework should associate local or remote semantics in a seamless manner.

How can components be connected to one another? Where is connection information stored?: Components establish communication links with each other via their typed ports. A *uses* port can be connected to a *provides* port by placing a handle to the *provides* port in a table that is local to the component that contains the *uses* port. When a method is invoked on the *uses* port, it can then be redirected to the *provides* port, by invoking it on the stored *provides* port. Each distributed framework needs to design its internal mechanism of obtaining a handle to the *provides* port and transporting it to an internal table of the component that has the *uses* port. Component connection information can be stored in two different ways: (1) it can be distributed across components, so that each component has information about the other components to which it is connected or (2) a central connection table can be maintained that keeps track of all the connections for an application. The latter approach facilitates the development of GUIs and other front-ends that show the component assembly of the application.

Where should the Builder Service be located? Do applications share a Builder Service, or does each component have its own?: The Builder Service provides specialized functionality to create and connect components. It could be designed as a well known stand-alone service that each component uses, or each component could have its own built-in Builder Service. This approach is useful if the Builder Service is lightweight, or if some components of the framework have special requirements and need the service to be specially tailored.

5 Case Studies

Our research focuses on the mappings from the CCA specification to distributed Grid frameworks. In this section we describe two distributed frameworks that have a binding for the CCA specification: Legion and XCAT. Each of the two subsections below is organized according to the requirements described in Section 4.

5.1 CCA over Legion

The Legion implementation of a distributed CCA framework primarily makes use of existing services and objects. We describe these services very briefly at a high level, but space considerations preclude detailed discussion, which can be found in other papers [6, 8]. In general, our approach to

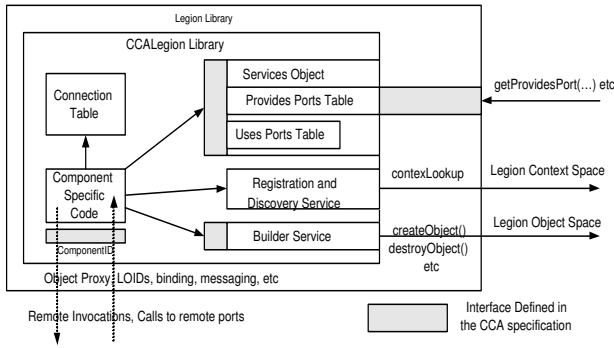


Figure 1. The anatomy of a Legion CCA component: Each CCA component is a Legion object which links both the Legion library and the CCALegion library. The CCA Legion library exposes the standard CCA specification interface to component code, and implements the functionality primarily by utilizing the facilities of the Legion library.

mapping the CCA specification to Legion, is to model CCA components as Legion objects. Legion objects that serve as CCA components are linked against a *LegionCCA library*, thereby giving them the added functionality to run within the CCA framework atop Legion. This library supports the API that is defined in the CCA Specification, thereby making its services and functionality available to applications programmers from within the components they build. To this end, the LegionCCA library contains implementations of important CCA types, including ComponentID and Ports, a Services Object and a builder service for the component, and a connection table that describes how the component is attached to others in the framework. The anatomy of a Legion CCA component is depicted in Figure 1, and the approach is further described in the remainder of this section below.

ComponentID: Legion’s CCA ComponentID type contains three data elements: *type*, *string name*, and *handle*. Currently, the handle is implemented as an LOID, which by definition serves as a unique identifier for the component within a Legion-based Grid. The string name is resolved within Legion’s *context space*—a global namespace of user-defined strings—and bound to the LOID of the Legion object that implements the specified component. Given the LOID, the Legion library can create a local proxy object within the calling component’s address space; the proxy is used to invoke the remote component’s functions.

The notion of serializability is called “packability” in the Legion library—it is the ability of a local C++ object to pack itself into a buffer for transfer to a remote Legion object. Context names and LOIDs are both packable objects, as implemented in the Legion library, and both types of names

have global scope. Therefore, serializing a ComponentID for a remote component is simply a matter of packing its data elements. To serialize a local component, only the context name needs to be packed (since the pointer to the local C++ object won’t necessarily be in within the scope of the remote location); the packed context name must then be used to re-bind the ComponentID and set the handle to the LOID of the Legion object in which the component resides.

Ports: The LegionCCA library supports a Services Object through which a component can manage both its *uses* and *provides* ports. The Services object contains two tables, one for each kind of port. Required ports are stored in the UsesTable, exported ports in the ProvidesTable. Access to the UsesTable can be restricted to local calls from within the component; that is, other remote objects need not know which ports a given component requires. The ProvidesTable requires both local and remote access—remote components must be able to query a component about the ports it implements. Functions for registering and removing *uses* and *provides* ports, as defined in the CCA specification, are implemented in the Services Object by manipulating the two tables appropriately.

Function identifiers and interfaces are both supported in Legion; Legion objects can retrieve a list of the functions exported by any other object in the system. Thus, the implementation of ports tables and serializable port names can make direct use of existing C++ classes (LegionFunctionID and LegionInterface) in the Legion library.

Remote and Local Access to CCA Functions: The CCA Specification provides the definitions for the functions that a framework must provide to component developers and CCA application builders. All of the functions must be made callable from component code, but some require remote implementations. For example, getting *provides* port information for a given component requires that the framework query the remote ProvidesPortsTable in the address space of the remote component. The approach taken in the LegionCCA library is to provide the full CCA-compliant interface in the library, and to map the implementations of the functions to remote service calls where necessary. Thus, syntactically, all CCA calls appear local, since the LegionCCA library contains a proxy for each of the CCA Specification’s defined interfaces (for Builder Service, the Services Object, the ComponentID, etc.).

Component Connections Component connections are maintained in a distributed manner across the components that are involved in the application. Each component contains a table that includes the information about which ports its *uses* parts are connected to, and which of its *provides* ports are bound to other components’ *uses* ports. This table can be updated dynamically as the component runs, to indicate connections that are made and broken as the application evolves. To make a connection, the handle for the remote

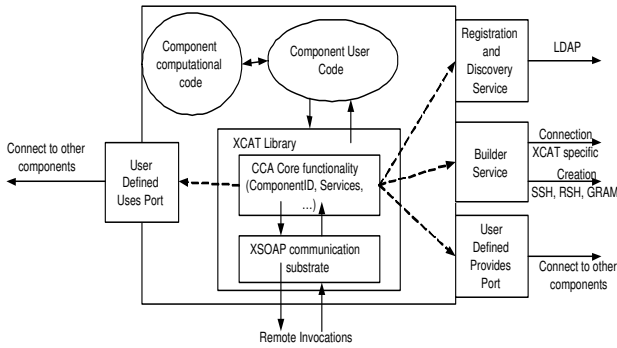


Figure 2. An example XCAT component: The ports reside within the XCAT library, but appear to component users as the interface of that component. XCAT provides Creation (via SSH, RSH, GRAM), Connection (XCAT specific) and Registration (via LDAP) to each component. Remote invocations are transmitted using the XSOAP system.

provides port is retrieved and placed in the field associated with the corresponding *uses* port in the client component. Likewise for the *provides* port when another component connects to a service exported by a component.

Builder Service Design The builder service, in charge of creating and destroying components, is supported by mapping its functions to the corresponding Legion functions. To instantiate a new component, the LegionLibrary’s *createObject(...)* function is called. To remove a component, *destroyObject(...)* is called. This simple mapping is possible because of the similarity between CCA Specification functionality and what is already provided in the Legion library, and because components are implemented directly as Legion objects.

5.2 CCA implementation with XCAT

The XCAT system has been implemented in Java and is designed to be compliant with both the CCA specification and Web Services standards. Figure 2 shows an example XCAT component with creation, connection, and registration bindings to specific protocols. XCAT uses XSOAP [10] for remote invocations of methods on ports of connected components. XSOAP is based on the Java-RMI design and uses the Simple Object Access Protocol (SOAP) as its wire protocol. XCAT uses the Globus GRAM mechanism (via the Java CoG kit [11]) for authenticated job creation on Grid resources. XCAT currently uses LDAP to register, search and retrieve components.

ComponentID: The ComponentID has been designed as a remote object. The format of the remote object is dependent on the XSOAP communication system. XSOAP uses an XML document (which is a subset of WSDL [3]) as its remote reference format. The ComponentID in its serialized form is an XML document. The namespace and other elements in the WSDL document uniquely identify the ComponentID in the distributed framework. When a method is invoked on a *uses* port, XCAT detects if the call is being directed to a co-located (same process) component, in which case it bypasses the expensive route of serialization and deserialization.

Ports: XCAT uses the factory design pattern to instantiate ports. Whenever a *uses* port type is registered (via a standard CCA call to the Services Object), the Services Object uses the factory to create a local object that is specialized for this type. However, when a *provides* port is registered via another standard CCA call to the Services Object), the Services Object converts the port into a remote object. This is done using the XSOAP library.

Remote and Local Access to CCA Functions: The Services Object has CCA methods that must be accessed by remote components, and some methods that can only be accessed locally. The Services Object implements a local and a remote interface. Access to the Services Object is controlled via these interfaces. All the parameters and return values in the remote interfaces are designed so that they can be serialized and deserialized.

Component Connections: The Builder Service has been designed as two separate modules, “Creation” and “Connection” services in XCAT. When the service is used to connect two components, XCAT uses non-CCA calls between components to retrieve the specific *provides* port and place it in the internal table of the component with the *uses* port. These calls are specific to the XCAT system and users are not made aware of these calls. XCAT does not maintain a central table to track all the connections in the application. Instead, XCAT uses an event system to send out events for creation, connection and other such activities. Interested parties can register as listeners and front-end tools can be built based on the event system.

Builder Service Design: XCAT has been designed as a services-based architecture. Grid services (creation, connection, registration, events, for example) are wrapped as CCA components so that users can directly interact with them. Each XCAT component has a Builder service associated with it. Consequently each component can create and connect other components. The services-based architecture allows the design of a layered system. For each layer it is possible to present users with a choice of implementation to use. For example, creation using the Builder Service can be done using “GRAM” or “SSH.”

6 Summary

We discussed the motivation for component based technology and discussed the important features of the Common Component Architecture (CCA) specification, which is gaining importance in the DOE community and universities. We focused on the design issues in mapping this specification to distributed Grid frameworks. We presented a list of choices that architects of distributed Grid frameworks need to consider. We showed how the CCA specification can be successfully used by two disparate Grid frameworks, Legion and XCAT. Performance is one of the key requirements for CCA components and computational Grids. In distributed frameworks performance is directly related to the the design decisions for the key characteristics discussed in Section 3. Among the characteristics mentioned in Section 3, the design choices for “remote invocation” are most important. We plan to conduct a performance analysis of our frameworks and study the trade-offs between high performance and interoperability for CCA based frameworks for the Grid.

References

- [1] B. A. Allan, R. C. Armstrong, A. P. Wolfe, J. Ray, D. E. Bernholdt, and J. A. Kohl. The CCA Core Specification In A Distributed Memory SPMD Framework. *CCPE*, 14(5):323–345, 2002.
- [2] CCA Forum. Common Component Architecture Forum, visited September, 2003. <http://www.ccaforum.org>.
- [3] E. Christensen et. al. Web Services Description Language (WSDL) 1.1, March 2001. <http://www.w3.org/TR/wsdl>.
- [4] I. Foster and C. Kesselman. *The GRID: Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 1998.
- [5] M. Govindaraju, S. Krishnan, K. Chiu, A. Slominski, D. Gannon, and R. Bramley. Merging the CCA Component Model with the OSGI Framework. In *3rd IEEE/ACM International Symposium on Cluster Computing and the Grid*, May 12-15, 2003, Tokyo, Japan.
- [6] A. Grimshaw, A. Ferrari, F. Knabe, and M. Humphrey. Legion: An operating system for wide-area computing. *IEEE Computer*, 32((5):(??)), 1999.
- [7] S. Kohn, G. Kurfert, J. Painter, and C. Ribbens. Divorcing Language Dependencies from a Scientific Software Library. In *Proceedings of 10th SIAM Conference on Parallel Processing*, Portsmouth, VA, March 12-14, 2001.
- [8] M. Lewis, A. Ferrari, M. Humphrey, J. Karpovich, M. Morgan, A. Natrajan, A. Nguyen-Tuong, G. Wasson, and A. Grimshaw. Support for extensibility and site autonomy in the legion grid system object model. *Journal of Parallel and Distributed Computing*, 63:(525–538), 2003.
- [9] Office of Science, Department of Energy. Top 10 DOE Science Achievements in 2002, visited October, 2003. <http://www.sc.doe.gov/sub/accomplishments/top10.htm>.
- [10] A. Slominski, M. Govindaraju, D. Gannon, and R. Bramley. Design of an XML based Interoperable RMI System : SoapRMI C++/Java 1.1. In *Proceedings of PDPTA*, pages 1661–1667, June 25-28, 2001.
- [11] G. von Laszewski, I. Foster, J. Gawor, and P. Lane. A Java Commodity Grid Kit. In *Concurrency and Computation: Practice and Experience*, vol. 13, no. 8-9, pp. 643-662, 2001.