XCAT3: A Framework for CCA Components as OGSA Services

Sriram Krishnan Dennis Gannon

Department of Computer Science, Indiana University.
215 Lindley Hall, 150 S Woodlawn Avenue, Bloomington, IN 47405-7104
{*srikrish, gannon*}*@cs.indiana.edu*

Abstract

The benefits of component technologies are well known: they enable encapsulation, modular construction of applications and software reuse. The DOE sponsored Common Component Architecture (CCA) [3] project adopts a component-based approach for building large scale scientific applications. On the other hand, the Web services-based Open Grid Service Architecture (OGSA) and Infrastructure (OGSI) [14] come close to defining a component architecture for the Grid. Using an approach where a CCA component is modeled as a set of Grid services, the XCAT3 framework allows for CCA components to be compatible with the OGSI specification. This enables CCA components to be accessible via standard Grid clients, especially the ones that are portal-based. For CCA compatibility, XCAT3 uses interfaces generated by the Babel [5] toolkit, and for OGSI compatibility, it uses the Extreme GSX [12] toolkit. In this paper, we describe our experience in implementing the XCAT3 system, and how it can be used to compose complex distributed applications on the Grid in a modular fashion.

Key Words: Grids, Components, Web Services, OGSA, OGSI, CCA, Babel, XSOAP, GSX.

1. Introduction

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies. A software component can be deployed independently and is subject to composition by third parties [20]. A component architecture is a system defining the rules of linking components together. The software engineering benefits of component based software are well known: they enable encapsulation, modular construction of applications and software reuse. Component systems are immensely useful to scientists who wish to build complex distributed applications by composing existing software components, thereby shielding them from the underlying complexity of the distributed set of resources.

Various component models have been successful in industry, as well as in academia. Microsoft's COM and DCOM frameworks have been fundamental to interoperability in Windows based applications. Their current Web services oriented .NET framework is also component based and is gaining widespread acceptance. In the CORBA world, the Object Management Group has released a specification for the CORBA Component Model (CCM) [4], whereas Java Beans and Enterprise Java Beans (EJB) [18] have been popular component standards for Java based applications. The CCA project, which is described here, is an initiative by DOE laboratories and universities to develop a common architecture for building large scale scientific applications from well-tested software components that run on both parallel and distributed systems. Several implementations of CCA exist, viz. XCAT [15], Ccaffeine [2], SCIRun [17].

A computational Grid [13] is a set of hardware and software resources that provide seamless, dependable, and pervasive access to high-end computational capabilities. Until recently, there has been no consensus on what programming model is appropriate for the Grid. The Open Grid Services Architecture (OGSA) is the first effort to standardize Grid functionality and produce a Grid programming model consistent with trends in the commercial sector. It integrates Grid and Web services concepts and technologies. The Open Grid Services Infrastructure (OGSI) refers to the basic infrastructure on which OGSA is built. At its core is the Grid Service Specification [11], which defines standard interfaces and behaviors of a Grid service in terms of Web services technologies. OGSA and OGSI come close to defining a component architecture for the Grid.

In our previous work [16], we argue for the need for making our CCA-based components compliant with Grid standards. Some of the arguments we make in the above paper are:

- By making our CCA components compliant with

OGSI, we can access them on the Grid using the same standards and clients used to access other Grid resources. This makes it easy to build Grid portals that can access and control our components on the Grid.

- The area of Web and Grid services is seeing a lot of research on standardization of various technologies such as Workflow (or composition in time), Service Level Agreements (SLAs), etc. These technologies should be leveraged to make components easier to use and more effective.
- Web and Grid services are lacking in standards that allow for application assembly (such as via component composition, which we refer to as composition in space). Concepts from standard component technologies such as CCA should be used to enable creation of complex applications on the Grid by composition of existing components.

However, merging the two standards (CCA and OGSI) is not a trivial task because of semantic differences between CCA components and Grid services which preclude a direct one-to-one mapping between components and services. Hence, in this paper, we present an approach where a component is modeled as a set of Grid services. We describe the design and implementation of XCAT3, which uses the above approach to create a framework that is compliant with both CCA and OGSI. With an example, we show how components written within the XCAT3 framework can interoperate and coexist with Grid services that are OGSI compliant, even if they may be written using other frameworks.

The rest of the paper is organized as follows. In Section 2, we describe the technologies that are vital for our project. In Section 3, we present the design and implementation issues that we have to deal with in order to make the XCAT3 framework compatible with both CCA and OGSI, and show how components can be written within our framework, and in Section 4, we discuss a typical scenario where XCAT3 can be used.

2. Relevant Technologies

In this section, we describe the Common Component Architecture (CCA) and the Open Grid Services Infrastructure (OGSI) in a little more detail. We also present the tools that we use for compatibility with these standards, viz. Babel for CCA compatibility, and Grid Service Extensions (GSX) for OGSI compatibility.

2.1. Common Component Architecture (CCA)

The CCA has primarily emphasized building applications and components for massively parallel supercomputers, but its semantics do not preclude its applicability to the Grid.

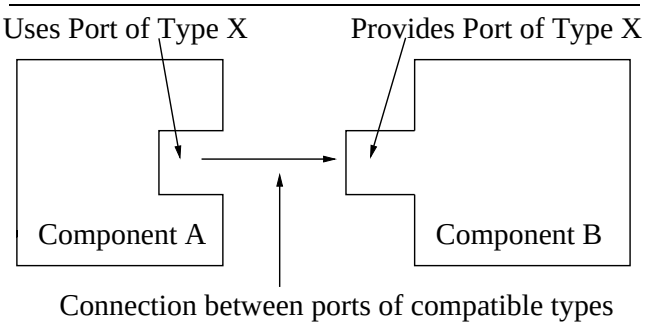

Figure 1. Example of a component connection using CCA. A uses port of type X can be connected to a provides port of the same type.

The central idea in CCA is to build applications by component composition. Two CCA components are composed by connecting together their *ports*. *Provides* ports represent functionality a component provides to other components. Semantically, these are almost similar to RPC Web service ports. *Uses* ports represent functionality a component may need. Uses ports are essentially bindable references to provides ports. After a uses port is connected to a provides port, any functionality represented by the uses port is obtained by invoking the connected provides port.

The CCA can be compared to the CORBA Component Model (CCM). Like the CCA, the CCM also has the notion of ports. The CCA uses port is analogous to the CCM receptacle, and the CCA provides port is analogous to the CCM facet. Unlike the CCM, however, the CCA envisions connections as a dynamic, run-time activity. Ports can be added, removed, and connected at run-time, and this is considered normal behavior. The CCM does not allow the addition or removal of ports. CCM connections are considered part of application assembly, and not something the end user would usually do dynamically. While the CCA also supports connections used in this manner, the more flexible nature of CCA ports and connections allow it to be used to as part of Problem Solving Environments (PSEs), in which the end-user directly manipulates component connections to solve the particular problem at hand.

Each port is identified by name and is described by an interface of operations. The interface can be described by the Scientific Interface Definition Language (SIDL) [8], or in our case by simple Java interfaces or by the Web Services Description Language (WSDL) [10]. Figure 1 shows an example of a connection between two components with compatible port types.

2.1.1. Babel. To define components and their interfaces, CCA uses the Scientific Interface Definition Language (SIDL) that was developed as part of the Babel project. SIDL addresses the unique needs of parallel scientific computing by supporting complex numbers and dynamic multi-dimensional arrays as well as parallel communication directives that are required for parallel distributed components. SIDL also provides other common features that are generally useful for software engineering, such as enumerated types, symbol versioning, name space management, and an object-oriented inheritance model similar to Java.

Babel tools parse interface descriptions for components in SIDL, and automatically generate glue code for the specified software library. This glue code mediates differences among calling languages and supports efficient inter-language calls within the same memory address space. In the future, Babel plans to support calls across memory spaces for distributed objects; however, this ability does not exist at this point.

2.2. Open Grid Services Infrastructure (OGSI)

The Open Grid Service Infrastructure extends the Web service model by defining a special set of service properties and behaviors. First, it separates the service naming and service reference. A Grid Service Reference (GSR) is a precise description of how to reach a service instance on the network. GSRs can be complete WSDL descriptions of a service instance. A Grid Service Handle (GSH), on the other hand, is an immutable name for a service. The idea is that a GSR may change over time as a service is moved or upgraded. Hence a GSH may be bound to different GSRs over time, but the GSH can always be resolved to the official version of the service instance.

The most important contribution of OGSI is the specification and restriction of Grid service behavior through the definition of a family of standard ports. The most important of these is the GridService port. This port provides dynamic service introspection, which is a common feature of component architectures. By invoking queries on the required Grid Service port, a client can discover information such as the other portTypes the service supports, the lifetime of the service instance, and other service-specific internal state data that the service wishes to expose. The information is conveyed back to the client in the form of XML fragments called Service Data Elements (SDEs). Each SDE is described by a Service Data Descriptor (SDD), which defines the schema and the content of the SDE.

Another important set of portTypes in OGSI involve notification. A client service can subscribe to changes in the service data of a source service by sending its GSH or GSR to the source service, via its NotificationSource portType. The notification source pushes SDEs back to the subscriber when they have changed. This is accomplished by invoking a DeliverNotification operation on the subscribing service. This provides a basic form of service composition.

2.2.1. XSOAP and Grid Service Extensions (GSX). XSOAP [19] (formerly called SoapRMI) is a lightweight implementation of Remote Method Invocation (RMI) that uses SOAP [9] as the communication protocol. XSOAP can also be used to write Web services, and any client that can understand the WSDL description for the service can make remote invocations on it.

GSX builds on top of XSOAP and provides extensions that enable users to expose their plain Web services as Grid services compliant with OGSI. The main additions to XSOAP are the OGSI specific portTypes that are required for a Web service to be a Grid service (e.g the GridService portType), the provision to add Service Data Elements to these services, and the use of multiple level naming (GSH and GSR, as described in Section 2.2).

3. The XCAT3 Framework

Currently, the XCAT3 framework is implemented in Java, and we plan to implement a corresponding C++ version that is inter-operable with the former.

3.1. General Architecture

Some of the key features of the XCAT3 architecture are as follows:

- *ComponentID*: Every component has a unique ComponentID that can be used to refer to it. CCA defines two operations for the ComponentID: `getInstanceName` which returns the name of the component instance, and `getSerialization` which returns the framework specific serialization of the ComponentID. However, we add a few methods to the *XCATComponentID* which is an interface that extends the ComponentID. These are used internally by the XCAT3 framework, especially for component composition purposes. The interesting ones are `getPortRef` to get a reference for a remote provides port that can be cached when a connection is made between a local uses and the remote provides port, `setPortRef` to cache a reference for a remote provides port when a connection is being made between a local uses and the remote provides port, `disconnectProvider` to disconnect the remote provides port from a connection with a local uses port (by removing the cached remote port reference), and `disconnectUser` to

notify a remote provides port that it has been disconnected from a uses port (for reference counting purposes).

- *Services*: Every component has a Services object that is set using the `setServices` method via the *Component* interface. The Services object is responsible for providing methods to register uses ports (`registerUsesPort`), add provides ports (`addProvidesPort`), fetch a previously register port (`getPort`), get a reference to the ComponentID (`getComponentID`), etc.

- *Exceptions*: CCA defines a set of exceptions that may occur during execution, viz. PortNotDefined, PortAlreadyDefined, PortNotConnected, BadPortName, etc. XCAT3 creates a Java Exception class for every CCA defined exception. All exceptions extend from the base *CCAException*. All exceptions thrown during communication between components are caught and returned to the component that initiated the communication. The exceptions are mapped to *SOAP faults* on the wire and then to corresponding exceptions on receiving end of the initiating component.

- *Builder Service*: The Builder service is a port implemented by CCA compliant frameworks for composing components into applications in a standard way. It exposes the component creation and composition functionalities. Some methods exposed by the Builder service for component lifecycle purposes are `createInstance` for creating an instance of a component, and `destroyInstance` for eliminating the component instance from the scope of the framework. Since ours is a distributed component framework, the createInstance is capable of creating component instances on remote locations using the Globus GRAM [7] protocol provided by the Java CoG Kit [21]. Upon successful creation of a component, a ComponentID is returned for the component which can be used to refer to it in the future. For composition purposes, the Builder services provides methods `connect` for connecting a uses port to a provides port, and `disconnect` for disconnecting an already existing connection.

- *Scripting Interface*: For rapid prototyping purposes, XCAT3 provides an interface to the Builder service using Jython scripts. Jython is pure Java implementation of the Python scripting language, and provides an almost seamless interface to code written in Java. Hence, exposing the functionality of the Builder service which was originally written in Java via Jython was a trivial task. The Jython API provided to the user closely mirrors the API provided by the Builder service.

3.2. CCA Compatibility

The CCA specification itself is defined in SIDL. The SIDL describes the various interfaces that a framework must implement in order to be compliant with CCA. It also defines the interfaces that ports and components must implement.

We use Babel to generate the Java interfaces from the SIDL specification in order to be compatible with CCA. However, Babel is currently defined for language interoperability within the same process space, and not for the distributed object case that we are interested in. Hence, we needed to strip out code from the stub classes that used Java Native Invocation (JNI) for native calls to other languages, and replace it with code that does remote invocations using XSOAP.

3.3. OGSI Compatibility

To be compatible with OGSI, we use the GSX toolkit to present both our provides ports, and our components as Grid services. We discuss each of the two cases separately in the following subsections.

3.3.1. Ports as Grid Services. As per the CCA specification, one component can have more than one instance of a provides port of the same type, where each instance is unique. As an example, consider the case of a remotely accessible electron microscope with multiple guns. In this case, the microscope itself will be represented as a component, while each of the guns will be represented as a provides port. In other words, these ports are envisioned to be state-ful.

On the other hand, in the Web services world (including OGSI), if a service has several ports of the same type (with different bindings and addresses), the ports are considered semantically equivalent. As a result, the same operations on different ports of the same type affects the state of the service in exactly the same way. Thus, Web service ports function as interfaces to the Web service, and can be inferred to be stateless.

Because of the above semantic difference between CCA provides ports and Web service ports, a model where every component is mapped to a Web (or Grid) service and every provides port is mapped to a Web service port is too restrictive, and is incorrect. Hence, we chose to model every provides port of a CCA component in XCAT3 as a separate lightweight Grid service (and hence, a Web service) using GSX. The XSOAP toolkit can be used to create the WSDL definition for these services and these can be published in registries such as UDDI [1] for interested clients.

Every provides port could conceivably be implemented as a simple Web service, and not a Grid service. However,

in the future, we plan to enable checkpointing and migration of XCAT3 components. The provision of multiple level naming for Grid services (and hence, XCAT provides ports) will help us keep our uses-provides connections location-independent, and help us in enabling migration for individual components. Additionally, the provides ports are now no different in behavior from other Grid services implemented using other frameworks such as the OGSI compliant Globus Toolkit (GT3). This means that a uses port can be connected to any OGSI compliant Grid service. Thus, existing Grid services that already provide functionalities that the users desire can be easily leveraged by simply connecting to them via uses ports.

In XCAT3, every port interface extends the *XCATPort* interface, which extends from *gov.cca.Port* (which is Babel-generated) and from *XSoapGridServiceInterface* which represents the OGSI GridService port, as provided by GSX. XCAT3 also provides a basic implementation of a provides port (*BasicPortImpl*) that every port implementation should extend from. This class implements the methods declared in the *XSoapGridServiceInterface*, obviating the need for the user to implement (or even be aware of) them. Thus, every provides port implemented by a user within the XCAT3 framework is automatically exposed as a Grid service.

3.3.2. Components as Grid Services. Every CCA component is a collection of provides ports, along with other shared state. It follows that every XCAT3 component is a collection of Grid services, which happen to be provides ports. We choose to expose every XCAT3 component itself as a Grid service, via its ComponentID.

The *XCATComponentID* implementation extends not only the Babel-generated *gov.cca.ComponentID*, but also the *XSoapGridServiceInterface* which makes it a Grid service. When a component is instantiated, the framework creates a *Container* for the instance. The container is responsible for managing the component instance. The container adds Service Data Elements to the *XCATComponentID* containing the names, GSH and GSR for the provides ports. Thus, any Grid client can obtain access to the provides ports by using this information from the *XCATComponentID*. In its current form, the container is very basic and performs limited functionality. But in the future, the container will be responsible for activities such as checkpointing the state of the component, evaluating performance guarantees or Service Level Agreements (SLA), etc.

Since the CCA ComponentID is an object that can be passed around so that any other user can have access to a particular component, we have to make sure that it can provide access to a remote component even if the component has migrated to another resource. In other words, the ComponentID implementation has to provide location-independence. Hence, the `getSerialization` method

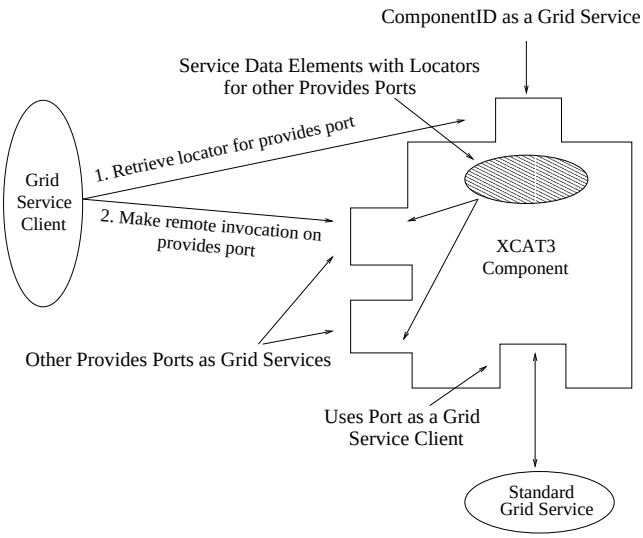

Figure 2. Every XCAT3 component is a Grid service. It contains SDEs with locators for all provides ports, which are also Grid services themselves

of the ComponentID in XCAT3 returns the GSH for the component that is valid for all time, and not the GSR which may no longer be valid if the component has migrated. The GSH along with a *HandleResolver* (which is an OGSI defined port that provides an ability to resolve a GSH into a valid GSR) is used by a client-side stub to make sure that the GSR for the remote component stays valid as long as the component is alive.

The modeling of an XCAT3 components and ports as Grid services is illustrated in Figure 2. The OGSI specification uses a concept called *ServiceGroups* to group together a set of Grid services that are related. We can model a component (using the ComponentID) as a ServiceGroup, and plan to do so in the future.

3.4. Writing Components

Writing components in the XCAT3 framework consists of writing the port interfaces, implementing the ports, writing the components, and writing scripts that create component instances and compose them meaningfully. We describe each of the above steps in this section.

- *Defining Port Interfaces*: In XCAT3, port interfaces are defined in Java. We don't use SIDL to describe our port interfaces because currently Babel tools have no notion of distributed objects. For any Java interface to function as a port in XCAT3, it needs to extend from *intf.ports.XCATPort*. As explained in Section 3.3, every port interface extends from *XCATPort* in order to

be compatible with the OGSI specification. The *XCATPort* (via its inheritance from *XSoapGridServiceInterface*) provides methods to query for Service Data Elements (SDE) and manage lifetime of the ports.

- *Implementing Ports*: XCAT3 provides a basic implementation of a port via class *xcat.ports.BasicPortImpl*. This class implements the methods present in the interface *XCATPort*, so that the user of the system need not be concerned with details of the OGSI specification. Hence, the user only needs to implement the methods that are added via the definition of the port interface.
- *Implementing Components*: Every implementation of a component has to implement the interface *gov.cca.Component*. The only method present in the *Component* interface is called `setServices` which is used to initialize the *Services* object for the component. Within the `setServices` method, a component is expected to create instances of provides ports and add them to the *Services* object using the `addProvidesPort` method, and also register uses ports using the `registerUsesPort` method. When a provides port is added, the XCAT3 *Services* implementation makes it available to the outside world as a Grid service transparently.
- *Scripting an application*: Once all the components have been written, the Builder service API can be used from Java to create instances of components, and compose them together to form a distributed application. However, as we mention in Section 3.1, Jython scripts can also be written for the same purpose. Currently, the Jython API to XCAT3 allows creating instances of components locally and remotely (using GRAM), connecting and disconnecting ports between component instances, querying Service Data Elements from components, destroying component instances, and invoking methods on provides ports contained within instantiated components.

In summary, Figure 3 describes the overall architecture of XCAT3, as described in the above sections.

4. Typical Scenario

Figure 4 illustrates a typical scenario in which the XCAT3 framework can be used. It shows how XCAT3 can be used to create a distributed application consisting of CCA components and OGSI services.

The application consists of a pool of *Worker* components, and a *Master* component that steers the computation with the aid of the Workers. Also present in the system is a *Registry* service that is OGSI compliant. The Registry is assumed to be persistent and implemented by a third party.

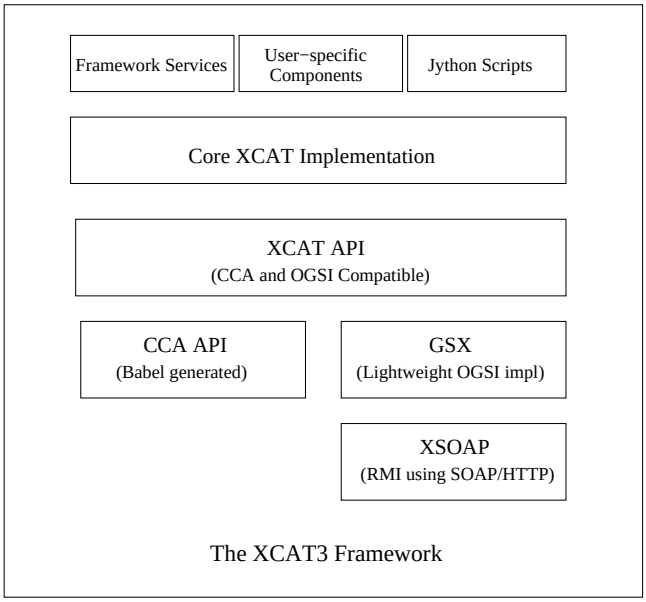

Figure 3. Architecture of XCAT3

Each of the Workers have a provides port that provides the core functionality of the Worker. The Master and the Workers communicate with the Registry with the help of uses ports. In addition, several uses ports are added dynamically to the Master for communicating with the Workers. And finally, the Master also has a *Builder* service port that is used to make connections between its uses ports and the provides ports of the Workers.

A Jython script is used to launch the Master and the pool of Worker components. The Jython script also connects the Master and the Workers with the Registry with the help of uses ports. Once the Workers are instantiated, they register *Locators* (GSH and/or GSR) to themselves with the Registry. In addition to the Locators, they also publish Service Data Elements periodically containing resource utilization and capabilities of their locations. In order to choose the best Worker for fastest turnaround time, the Master queries the Registry for Workers with the best availabilties and capabilities, and receives the Locators for the same. It then uses its Builder service to make a connection with the chosen Worker, and sends out a work packet via the uses port.

Thus, XCAT3 can be used to build a complex distributed application by composition of regular CCA components, along with Grid services as long as they are compatible with OGSI. Even though such a system could conceivably be built using vanilla distributed CCA components or plain Grid services, a combination of these models helps us use concepts of component composition (via the uses-provides mechanism) for modular construction of complex applications in the realm of Grid services. Additionally, the compliance with OGSI opens up avenues for reusing hundreds of

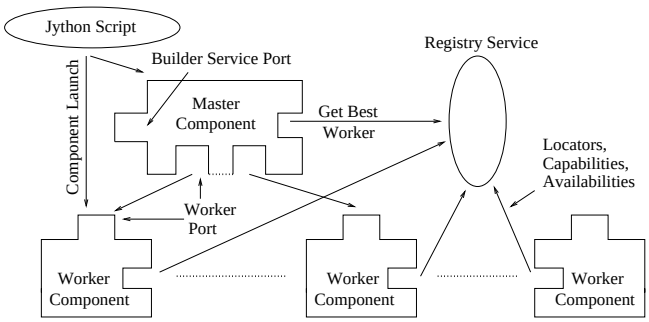

Figure 4. Typical scenario for the use of XCAT3

Grid services and clients available in the fast growing Grid community (e.g the Registry service that is used in our scenario).

5. Conclusions

We have presented a distributed software component framework, XCAT3, that is compatible with the Common Component Architecture (CCA) and also compliant with the Open Grid Services Infrastructure (OGSI). This allows the combination of benefits from both models - the ability to construct complex distributed applications by composition due to the former, and the ease of use in a Grid-based environment due to the latter.

CCA components have two types of *ports*: *Provides* which is identical to a Web service port, and *Uses* which is an external reference from one component to a provides port of another component. Provides ports are modeled as OGSI services in XCAT3, and the component itself is modeled as an OGSI service consisting of a set of OGSI-compliant provides ports. The Grid Service Handles (GSH) and References (GSR) to the ports contained by the component can be obtained from the introspection information available from the component, which is represented as Service Data Elements (SDE).

In the future, we plan to implement a C++ version of XCAT3 that is compatible with the Java version. The C++ version will use the Proteus Multi-protocol library [6] for better performance for scientific applications.

6. Acknowledgements

This work is supported by National Science Foundation grants EIA-0202048 and NSF-0116050, and the Department of Energy Office of Science SciDAC grants. We also wish to acknowledge insights from our colleagues at the Extreme Lab at Indiana University, in specific Randall Bramley, Kenneth Chiu, Madhusudhan Govindaraju, and Aleksander Slominski.

References

[1] Universal Description Discovery and Integration of Business for the Web (UDDI), Dec 2003. http://www.uddi.org/specification.html.

[2] 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. *Concurrency and Computation: Practice and Experience 14(5)*, 2002.

[3] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a Common Component Architecture for High-Performance Scientific Computing. In *8th IEEE International Symposium on High Performance Distributed Computation*, August 1999.

[4] D. Bartlett. CORBA Component Model (CCM): Introducing next generation CORBA, Dec 2003. http://www-106.ibm.com/developerworks/webservices/library/co-cjct6.

[5] Center for Applied Scientific Computing (CASC), LLNL. The Babel Project, Dec 2003. http://www.llnl.gov/CASC/components/babel.html.

[6] K. Chiu, M. Govindaraju, and D. Gannon. The Proteus Multiprotocol Library. In *Supercomputing 2002*, November 2002.

[7] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In *IPPS/SPDP 98, Workshop on Job Scheduling Strategies for Parallel Processing*, 1998.

[8] N. Elliott, S. Kohn, and B. Smolinski. Language Interoperability for High-Performance Parallel Scientific Components. In *International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE 1999)*, September 29 - October 2 1999. San Francisco, CA.

[9] D. Box et al. Simple Object Access Protocol 1.1, Dec 2003. http://www.w3.org/TR/SOAP.

[10] E. Christensen et al. Web Services Description Language (WSDL) 1.1, Dec 2003. http://www.w3.org/TR/wsdl.

[11] S. Tuecke et al. Grid Service Specification, April 2003. http://www.gridforum.org/ogsi-wg/drafts/draft-ggf-ogsi-gridservice-29_2003-04-05.pdf.

[12] Indiana University Extreme Computing Lab. Grid Service Extensions (GSX), Dec 2003. http://www.extreme.indiana.edu/xgws/GSX.

[13] I. Foster and C. Kesselman. *The GRID: Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 1998.

[14] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid Services for Distributed System Integration. *Computer 35(6)*, 2002.

[15] M. Govindaraju, S. Krishnan, K. Chiu, A. Slominski, D. Gannon, and R. Bramley. XCAT 2.0: Design and Implementation of Component based Web Services. Technical report, Department of Computer Science, Indiana University, June 2002. TR562.

[16] M. Govindaraju, S. Krishnan, K. Chiu, A. Slominski, D. Gannon, and R. Bramley. Merging the CCA Component Model with the OGSI Framework. In *3rd IEEE/ACM International Symposium on Cluster Computing and the Grid*, May 2003.

[17] C. Johnson and S. Parker. The SCIRun Parallel Scientific Computing Problem Solving Environment. In *9th SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

[18] R. Monson-Haefel. *Enterprise Java Beans*. O'Reilly, 1999.

[19] A. Slominski, M. Govindaraju, D. Gannon, and R. Bramley. Design of an XML based Interoperable RMI System : SoapRMI C++/Java 1.1. In *International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, Pages 1661-1667*, June 25-28 2001.

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

[21] G. von Laszewski, J. Gawor, S. Krishnan, and K. Jackson. *Grid Computing: Making the Global Infrastructure a Reality*, chapter 25, Commodity Grid Kits - Middleware for Building Grid Computing Environments. Wiley, 2003.