Merging the CCA Component Model with the OGSI Framework

Madhusudhan Govindaraju, Sriram Krishnan, Kenneth Chiu,
Aleksander Slominski, Dennis Gannon, Randall Bramley

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

Abstract

The most important recent development in Grid systems is the adoption of the Web Services model as its basic architecture. The result is called the Open Grid Services Architecture (OGSA). This paper describes a component framework for distributed Grid applications that is consistent with that model. The framework, called XCAT, is based on the U.S. Department of Energy Common Component Architecture (CCA) but with an implementation based on the standard Web Services stack. Using this framework, an application programmer can compose an application from a set of distributed components. The result is a set of Web Services that collectively represent the executing application instance. This paper describes the basic architecture of XCAT and the design issues to be considered for a component to serve as both a CCA and Open Grid Service Infrastructure (OGSI) service.

Key Words: Computational Grids, Component Architectures, Web Services, OGSA, OGSI, CCA, Composition, Workflow

1 Introduction

A computational Grid [17] is a set of hardware and software resources that provide seamless, dependable, and pervasive access to high-end computational capabilities. By enabling the use of teraflop computers and petabyte storage systems interconnected by gigabit networks, the Grid will allow scientists to explore new avenues of research. The success of the Grid depends largely upon the development of tools and applications that can exploit its potential, and make it easy for the end user to use them.

A programming model for the Grid consists of tools, conventions, protocols, language constructs, and a set of libraries that encapsulate important functionality. The abstractions provided by the programming model can simplify development of complex Grid applications. Until recently, there has been no consensus on what programming model is appropriate for the Grid. Examples of various models currently in use include MPI [16] for message passing, and GridRPC [27] for doing remote procedure calls.

The Open Grid Services Architecture (OGSA) [18] 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 [15], 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.

A component architecture can be defined as a set of rules for specifying the behavior and interfaces of component instances and a framework that allows the component to be composed into applications. A component is a software object or process that satisfies the rules of the architecture. The software engineering benefits of component based software are well known: they enable encapsulation and facilitate the modular construction of programs and the reuse of existing components, resulting in improved application productivity. Component architectures are well-suited for the rapid prototyping of complex distributed applications. These systems are immensely useful to scientists who build applications by composing existing software components which exploit specialized computing and algorithmic resources, and hold great promise as an effective programming model for the Grid.

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 importance. 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) have been popular component standards for Java based applications. The CCA [3] 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.

This paper presents three significant contributions to Grid research. The first is the merging of the CCA with the OGSI through our work with XCAT, which is our implementation of the CCA specification. A unification of the two models benefits both efforts, yet lets each group focus on their particular needs. Since the two specifications are evolving, our work on merging the two specifications has focused on identifying OGSI concepts that directly map to CCA features.

Perhaps the most important aspect of using a component Grid architecture is the way components are composed to construct the application. Our second contribution is our classification of composition modalities based on their extents in space and time. We observe that distributed systems may be composed in two ways:

- Composition in space: one component/service directly invokes the services of another component through a logical connection between the two.
- Composition in time: a workflow engine schedules tasks that involve accessing remote services and responding to events.

We argue that both of these are important and they can be accommodated in the XCAT framework.

Our third contribution is a messaging and notification system that extends the model proposed by OGSI. We also briefly describe how the OGSI factory service can be extended and used in applications.

2 A Brief Tour of Web Services

A Web Service is an interface to application functionality that is accessible using well-known Internet standards and is independent of any operating system or programming language. Web Services represent a shift in paradigm from a human-centric to an application-centric web.

The various protocols composing a Web Service are commonly divided into a five-layer stack as shown in Figure 1. This stack is evolving as various groups refine the standards.

Stack Layer	*Example Technologies*
Framework	.NET, Sun ONE
Discovery	UDDI
Description	WSDL, RDF
Messaging	SOAP
Transport	HTTP, SMTP, FTP, BEEP

Figure 1. Different layers of the Web Service stack and the example technologies for each layer.

1. **Transport**: The transport layer refers to the technology responsible for transferring messages between applications. The choices for this layer include HTTP, SMTP, FTP, and BEEP [1].
2. **Messaging**: This layer represents the marshaling and unmarshaling of application data so that it can moved over the network. Even though HTML has been widely used for the Web, it is not a suitable format for marshaling because it only describes the presentation of data, and not its semantics. XML, on the other hand, has gained widespread acceptance for representing data for Web Services as it allows for a representation in accordance with the meaning of the data. SOAP is a protocol that uses XML as its data format and is the *de facto* standard for messaging in Web Services.
3. **Description**: The description of a Web Service includes the supported interface, network, transport and packaging protocols. The Web Service Description Language (WSDL) [11] is a widely accepted standard for this purpose. The Resource Description Framework (RDF) [24] specification can also be used, though it is less popular than WSDL.
4. **Discovery**: This layer serves as a registry that enables Web Services to be published and discovered. The most widely recognized mechanism for this purpose is the Universal Description, Discovery, and Integration (UDDI) [2] specification.
5. **Framework**: The framework layer defines conventions and higher-level services that are important for an intended class of applications. These standards make it easier for applications to interoperate. Examples of such frameworks include Microsoft's .NET and Sun Open Net Environment (ONE) [26].

In WSDL terms, a Service is a collection of *ports*. Each port is a named association between a *binding* and some form of network address. A binding is an association of a *portType* with a set of protocols and message formats. A portType defines a set of *operations*, which are defined by the operation name and the *types* of the input, output and fault *messages* that are associated with the operation. A WSDL description of a service is an XML document that defines the types, messages, portTypes, bindings and ports associated with a service.

3 The Open Grid Services Infrastructure

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 Grid Service 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, but as we will argue below, it is not sufficient for a wide range of Grid applications.[1]

4 The Common Component Architecture

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

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 similar to simple RPC Web Service

[1]The Global Grid Forum (GGF) OGSI working group is currently considering extensions to this notifi cation model, so it may be changed by the time this article appears.

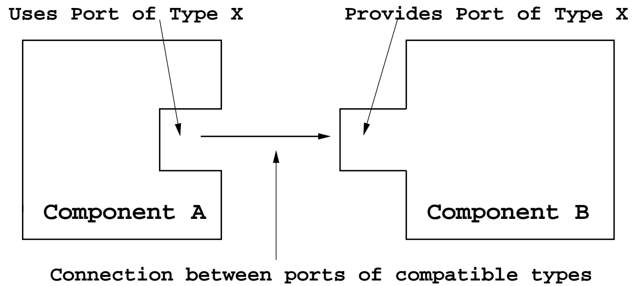

Figure 2. Example of a component connection using CCA. A uses port of type X can be connected to a provides port of the same type.

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 a simple Java interface, or by an XML specification such as WSDL. Figure 2 shows an example of a connection between two components with compatible port types.

Like OGSA, CCA provides a standard set of services and ports. The most important of these are the component creation service, which allows one component instance to create an instance of another component, and the connection service, which allows the programmer to bind a uses port in one component to a provides port in another component. The CCA community is working on combining these two services into a single service named the *Builder Service*.

5 XCAT: A Web Service-Based CCA Implementation

In our previous work [5] we presented an implementation of the CCA specification. It was primarily built as a research vehicle to test the viability of CCA for distributed computing. The system was built using HPC++ [20] and NexusRMI [6] as the underlying communication layer. The binary format of the communication substrate was ill-suited for exposing components as Web Services. We redesigned and implemented the second version (now called XCAT 2.0) with SOAP [9] as the communication protocol. XCAT focuses on leveraging the advantages of both the component and Web Services world. It has been implemented in both C++ and Java, and it provides seamless interoperability between components written in these two languages.

Every provides port in the XCAT implementation is a Web Service with one portType. The port interfaces are described using XML documents conforming to a schema that has a subset of the features in WSDL[2]. These documents are also used to generate the wrapper code that shields the users from the low-level details of the communication substrate used by XCAT. The generated code also handles the required conversion for seamless interoperability between C++ and Java based components.

XCAT uses the XSOAP [28] communication system for messaging, which provides an elegant model for communication between objects in different address spaces. XSOAP (formerly called SoapRMI) is an implementation of the Java RMI model in Java (XSOAP-Java) and C++ (XSOAP-C++) that uses SOAP as the communication protocol. XSOAP-Java uses the dynamic proxy feature, introduced in Java 1.3, to dynamically generate stubs and skeletons for every remote method invocation. Since C++ does not have introspection capabilities, XSOAP-C++ uses statically generated stubs and skeletons. We are currently working on porting the Proteus Multiprotocol Library [7] to XCAT. This will give us the option of using a multitude of communication libraries that include SOAP, JMS [25] and binary protocols.

XCAT provides a Creation service that encapsulates the component instantiation mechanism, thus shielding the component developers from the low-level, implementation-specific details of the instantiation mechanisms. This service allows:

1. Creating instances of components from a set of environment values, such as executable location, host machine, and creation mechanism. A new component can be created in the same address space as the creating component or it may be instantiated in a different one on another host, in which case Globus GRAM (via the Java CoG Kit [31]), or ssh (if no queuing is desired, and Globus is unavailable) can be used. Upon successful instantiation of the component, the creation service returns a ComponentID that serves as a handle for the new component.
2. Deleting instances of components by using their ComponentIDs.

[2]We are currently in the process of moving to full fledged WSDL for this purpose.

XCAT also provides a Connection service which allows instantiated components to establish communication links with one another via their typed ports. By providing an external mechanism for connecting ports, the port types and descriptions themselves can remain free of any connection semantics. This service allows:

1. Connection and disconnection between ports. This facilitates dynamically changing the application composition.
2. Exporting the ports of another component as one's own. Using this feature, wrapper components can expose selected functionality of other components. This allows a component to present a simplified interface to the end-users, shielding them from the unnecessary details.

Other features of XCAT that are important for its use as a distributed computing framework are enumerated below.

1. Security: Every remote method call can be intercepted by the XCAT-Java framework before it invokes a method on the provides port. A security service can thus be interposed between the provides port and the XCAT framework. This security service can inspect the call and allow its passage if the security requirements have been met. The current version uses SSL with X.509 certificates and supports both authentication as well as a simple authorization model based on access control lists. This has been discussed in detail in [13].
2. ComponentID: The ComponentID represents a transportable handle to the component. XCAT uses the remote reference mechanisms provided by XSOAP to represent a ComponentID. This handle can be *stringified* and published to a registry service. It can then be discovered by interested parties and used to invoke methods on the component. The ComponentID in this serialized form is an XML document that describes the component. This XML document can be converted to a WSDL document using a tool provided by the XSOAP toolkit.
3. Exceptions: XCAT provides an exception model for communication between components. 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.

4. Events and Notification: XCAT-Java uses an event notification system called XMessages [29], which is a messaging middleware system designed for Grid applications that reliably delivers XML messages from publishers to subscribers even if a subscriber moves to a new location, or a publisher is restarted.

6 XCAT and OGSI

The CCA and OGSI specification share many design features. However, there are also some differences. In this section we discuss issues involved in mapping the requirements of OGSI into the XCAT framework.

1. In the CCA world, a component can have more than one instance of a port of the same type, where each instance is unique. However, in the Web Services world, if a service has several ports of the same type (with possibly different bindings or addresses), the ports provide *semantically equivalent* behavior. As a result, the same operation on different ports of the same type affects the state of the service in *exactly* the same way. This is not true for CCA components. Hence, each provides port of an XCAT component is mapped to a separate Web Service, so that they can be uniquely identified and need not exhibit semantically equivalent behavior.

   Each XCAT component has a provides port with the standard OGSI Grid Service functionality. SDEs containing references to all provides ports of the component are added to the Grid Service port, thus providing a list of services (provides ports) that can be accessed.

2. The ComponentID of a CCA based component uniquely identifies the component. It can be represented as a string and each CCA implementation can convert it to its native format. For XCAT, the string representation functions as the GSH, while the WSDL document functions as the GSR. However, in XCAT, the GSH does not refer to a particular provides port of the component, but to the entire component. In OGSI, the functionality for mapping a GSH to a GSR is provided by the `HandleResolver` service. By default, this mapping is internal to XCAT and can be made available as an optional service.

   In the OGSI specification, a GSH is required to be an URI. However, in CCA implementations there is no restriction on the format of the string representation of a ComponentID.

3. Most component specifications provide mechanisms for managing the lifecycle of components. In OGSI, lifecycle management is handled by the `CreateService` operation on a `Factory` service, and the `Destroy` operation on the Grid Service port of the service. In XCAT, this functionality is provided by the Creation service, which can be easily modified to invoke the `CreateService` operation on a `Factory`, and the `Destroy` operation on the component's Grid Service port.

4. OGSI messaging, in its current form, is based on a very simple, point-to-point, non-reliable event *push* model. Messages result only from changes in defined service data. However, Grid Services may wish to communicate more than just simple service data between themselves asynchronously. The use of XMessages in XCAT-Java provides a network of reliable message channels which store messages until they expire or they are explicitly removed from the system. It is possible to retrieve historical messages as the channel uses a persistent store. A message can be an arbitrary XML fragment such as an OGSI SDE. In addition to having messages *pushed* to clients, a client can *pull* messages from the channel in batches. This makes the client more mobile and firewall-friendly. XCAT provides a richer model for messaging and notification than OGSI.

5. OGSI has a Factory portType for instantiating services. Using XCAT, we have implemented an extended version of a distributed factory model for creating instances of applications [19]. Our generic application factory service takes a description of a connected network of components as input and creates an application coordinator component instance which, in turn, creates and links together instances of the described network of components. A reference to the new manager instance is returned to the client of the factory service.

Our current implementation does not conform to OGSI in some small details. This is because the specification is still being developed. However, we consider OGSI to be important and will continue to work towards an implementation that fully merges it with the CCA standard.

7 Composition in Space: Component Assembly

Using a component architecture requires describing the various components that constitute an application along

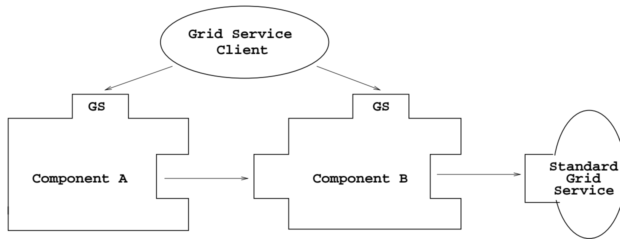

Figure 3. Each XCAT component has one provides port that implements the OGSI Grid Service Port Type. All other provides ports are first-class Web service ports as well as CCA ports. Components can be connected using CCA connection primitives and XCAT uses ports can be connected to conventional Web Services.

with the interconnections between them. In this model which we call, composition in space, component instances are created on specified hosts and then connected together as a distributed system. It is also possible to create *meta-components* which are themselves created by composing a number of components together.

To accomplish this kind of composition, the XCAT Services APIs can be used directly by the user to write simple Java programs that can use remote component instances. The Java program can use the Creation Service to create components and obtain references to running instances. The program can then use the Connection Service to connect the provides and uses ports of these components. The Naming Service can be used to store and retrieve handles to running instances of components. It is also possible to invoke specific methods on the ports of various components.

The above method (of using Java control programs) is only suitable when we have a fixed set of components which need to be launched and monitored. A more dynamic mechanism to create and manage components on the fly, without the need for any recompilation, is desirable. We use Jython scripts for this purpose. Jython is a pure Java implementation of the Python language. Since XCAT has an implementation in Java, we can provide a Jython interface to the XCAT libraries.

Apart from the above two ways to orchestrate computations (Java control programs and Jython scripts), application-specific GUIs can be easily written and layered on top of the services provided by XCAT.

As illustrated in Figure 3, we can combine component instances together using the XCAT composition to form composite Grid Services which may be accessed by any Grid Service client.

8 Composition in Time: Workflow

One of the most compelling reasons for the acceptance of the Web service technologies is their ability to combine existing processes and services into new ones that are more useful. *Workflow* can be defined as an organization of processes into a well-defined flow of operations, and can be thought of as the composition of services over time to accomplish a specific goal. This is one area that well surpasses the current OGSI specification.

While the composition of components in space defines how the components are logically connected at any point of time, workflow systems define ways in which flow of control and data can be expressed. As an example, an activity X in a workflow system may involve invoking operations P on service A, Q on service B, and R on service C (in that order), while service C may itself be composed of components D and E in space. Thus, these compositions are orthogonal, and can be applicable at the same time.

Currently, workflow systems for Grid and Web Services are evoking a high degree of interest, with projects such as WSFL [23], BPEL4WS [12], and GSFL [22] investigating the various aspects of workflow in their respective domains. However, most workflow systems for Web Services do not effectively address composition in space, because WSDL 1.1 does not completely define *outgoing* operations, i.e. where the Web Service itself is the caller. However, XCAT combines the benefits of a standard component architecture that uses component assembly as a mechanism to provide composition in space, and that of a Web/Grid Services architecture that uses a workflow specification to provide composition in time. Hence, we can leverage and apply the work done in component assembly and workflow systems to design a *meta-composition* system that will provide both composition in space and time, which is not very easily done if only one of these architectures is applicable.

9 XCAT Applications

The XCAT Science Portal [14] uses the XCAT implementation as the underlying model for launching distributed applications on the grid. Some projects that use the XCAT system are IU Xports [14], NCSA's Weather Research and Forecasting, (WRF) and Chemical Engineering [21], GRAPPA [30], Collision Risk Assessment (CRASS) [10], and Linear Systems Analysis (LSA) [10]. To illustrate the use of XCAT for composition of components in state and space, we show a typical scenario in Figure 4, which is seen in applications mentioned above, such as CRASS and Chemical Engineering.

As shown in the figure, the whole application is steered by an Application Coordinator, which is responsible for the

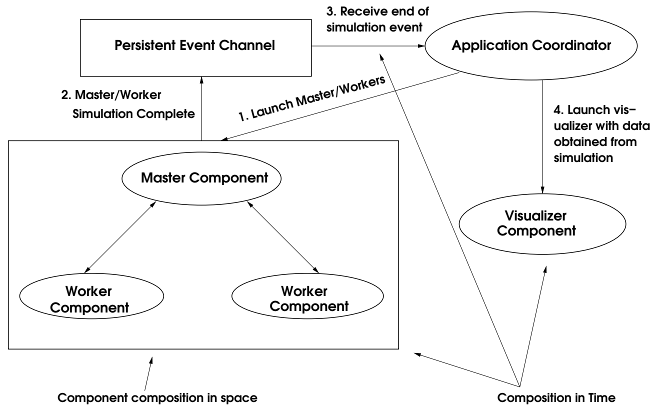

Figure 4. A scenario illustrating component composition in space and time using XCAT.

composition of the other XCAT components. The Application Coordinator first launches a Master component and a set of Worker components, and makes appropriate connections between them. The Application Coordinator then subscribes to the Event Channel in order to receive a *Simulation Complete* Event from the Master component. When the simulation is complete, the Master sends the data from the simulation as an asynchronous event to the Event Channel. The Event Channel stores this data in persistent storage (for possible future use), and then relays it to the Application Coordinator. On receiving this event from the channel, the Coordinator spawns a local Visualizer, sending it the data just received. The Visualizer then visualizes the results from the simulation. Thus, XCAT enables composition in space (Master and Worker components), as well as in time (Master/Worker and Visualizer components), so as to allow orchestration of complex distributed applications.

10 Conclusions

We have presented a distributed software component architecture, XCAT, for Grid computing that is compatible with the Common Component Architecture specification (CCA). We also discuss our work on merging this specification with the Open Grid Service Infrastructure (OGSI). The two specifications are still evolving and we plan to continue working towards building a system that unifies the two models.

CCA components have two types of *ports*. One type of port, which is called a provides port, is identical to a Web Service port. The other type, called a uses port, is an external reference from one component to a provides port on another component that can be bound at runtime. XCAT can also make use of an XML-based messaging system that provides a simple way for components to publish or subscribe to messages. This message system is substantially richer than the current OGSI notification scheme.

We have argued that the process of building distributed applications can be accomplished by either composing a collection of concurrently running components by linking their uses and provides ports (composition in space), or by scheduling the workflow between components and synchronizing activities based on the publication of application specific events (composition in time). There will be many cases when both schemes can be used together, and XCAT provides a mechanism to do so.

References

[1] The Blocks Extensible Exchange Protocol Core (BEEP), March 2001. http://www.ietf.org/rfc/rfc3080.txt.

[2] Universal Description Discovery and Integration of Business for the Web (UDDI), September 2000. http://www.uddi.org/specification.html.

[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 *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computation*, August 1999.

[4] D. Bartlett. CORBA Component Model (CCM): Introducing next generation CORBA, April 2001. http://www-106.ibm.com/developerworks/webservices/library/co-cjct6.

[5] R. Bramley, K. Chiu, S. Diwan, D. Gannon, M. Govindaraju, N. Mukhi, B. Temko, and M. Yechuri. A Component Based Services Architecture for Building Distributed Applications. In *Proceedings of Ninth IEEE International Symposium on High Performance Distributed Computing Conference*, August 2000.

[6] F. Breg, S. Diwan, J. Villacis, J. Balasubramanian, E. Akman, and D. Gannon. Java RMI Performance and Object Model Interoperability: Experiments with Java/HPC++. *Concurrency and Experience*, 1998.

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

[8] N. Elliott, S. Kohn, and B. Smolinski. Language Interoperability for High-Performance Parallel Scientific

Components. In *Proceedings of 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, May 2000. http://www.w3.org/TR/SOAP.

[10] D. Gannon et al. Programming the Grid: Distributed Software Components, P2P and Grid Web Services for Scientific Applications. In *Special Issue on Grid Computing, Journal of Cluster Computing*, July 2002.

[11] E. Christensen et al. Web Services Description Language (WSDL) 1.1, March 2001. http://www.w3.org/TR/wsdl.

[12] F. Curbera et al. Business Process Execution Language for Web Services, Version 1.0, July 2002. http://www-106.ibm.com/developerworks/library/ws-bpel.

[13] L. Ramakrishnan et al. An Authorization Framework for a Grid Based Common Component Architecture. In *Proceedings of the 3rd International Workshop on Grid Computing*, November 2002.

[14] S. Krishnan et al. The XCAT Science Portal. In *Proceedings of Supercomputing 2001*, November 2001.

[15] S. Tuecke et al. Grid Service Specification, October 2002. http://www.gridforum.org/ogsi-wg/drafts/draft-ggf-ogsi-gridservice-04_2002-10-04.pdf.

[16] I. Foster and N. Karonis. A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems. In *Proceedings of Supercomputing 1998*, November 1998.

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

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

[19] D. Gannon, R. Ananthakrishnan, S. Krishnan, M. Govindaraju, L. Ramakrishnan, and A. Slominski. *Grid Computing: Making the Global Infrastructure a Reality*, chapter 9, Grid Web Services and Application Factories. Wiley, 2003.

[20] D. Gannon, P. Beckman, E. Johnson, and T. Green. *Compilation Issues on Distributed Memory Systems*, chapter 3 HPC++ and the HPC++Lib Toolkit. Springer-Verlag, 1997.

[21] 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.

[22] S. Krishnan, P. Wagstrom, and G. von Laszewski. GSFL: A Workflow Framework for Grid Services. In *Argonne National Laboratory, Preprint ANL/MCS-P980-0802*, August 2002.

[23] F. Leymann. Web Services Flow Language (WSFL 1.0), May 2001. www-4.ibm.com/-software/solutions/webservices/pdf/WSFL.pdf.

[24] F. Manola and E. Miller. RDF Primer, January 2003. http://www.w3.org/TR/rdf-primer.

[25] Sun Microsystems. Java Messaging Service, March 2003. http://java.sun.com/products/jms.

[26] SUN Microsystems. Sun Open Net Environment, March 2003. http://www.sun.com/sunone.

[27] H. Nakada, S. Matsuoka, K. Seymour, J. Dongarra, C. Lee, and H. Casanova. GridRPC : A Remote Procedure Call API for Grid Computing, July 2002. www.gridforum.org/Meetings/ggf5/pdf/APM11.pdf.

[28] 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 the International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, Pages 1661-1667*, June 25-28 2001.

[29] A. Slominski, Y. Simmhan, A. Rossi, M. Farrallee, and D. Gannon. XEvents/XMessages: Application Events and Messaging Framework for Grid, October 2002. http://www.extreme.indiana.edu/xgws/papers/xevents-_xmessages_tr.

[30] Indiana University. Grid Access Portal for Physics Applications, March 2003. http://iuatlas.physics.indiana.edu/grappa.

[31] 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.