XCAT 2.0: Design and Implementation of Component based Web Services

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

Department of Computer Science, 150 S Woodlawn Avenue.
Indiana University, Bloomington, IN 47405. Phone: (812) 855-8305.

Abstract

The most important recent development in Grid systems is the adoption of the web services model as a basic architecture for Grid services. This paper describes a component framework for building 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 but with an implementation based on the standard web services stack. Using this framework, it is possible for an application programmer to build distributed applications by composing software components running on remote resources. The result is a transient, stateful web service that represents the executing application instance. This paper describes the basic architecture of XCAT and a programming/scripting model for building the composed application services.

Key Words: Computational Grids, Component Architectures, Web Services

1 Introduction

A computational Grid [10] is a set of hardware and software resources that provide seamless, dependable and pervasive access to high-end computational capabilities. The Grid has the potential to provide programmers with the capability to explore a new generation of interesting applications that can leverage teraflop computers and petabyte storage systems interconnected by gigabit networks. Given the heterogeneous nature of the computing environment on the Grid, programmers have to be concerned with many low level details. However, just as most computer users today do not have to write programs, most end users of the Grid should only utilize grid-enabled applications. The success of the Grid will largely depend 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 a useful functionality. A high level programming language paradigm that can shield the users from the low level details of each resource is the key to building effective applications for the Grid. The abstractions provided by the programming model can simplify development of complex Grid applications. The design of a standard programming model will also result in division of labor between users and developers of the various parts of an application. Currently, there is no consensus on what programming model is appropriate for the Grid. Examples of various models currently in use include MPI [32] for message passing, Condor [35] for high throughput, and HPF [29] for data parallelism.

The software engineering benefits of component based software are well known: they enable encapsulation and facilitate in the modular construction of programs and the reuse of existing components, resulting in improved application productivity. Component architectures are well suited for rapid prototyping of complex and distributed applications. They provide a natural way to incorporate existing scientific software code base as components to build applications. These systems are of immense utility to scientists who want to build applications by composing existing software components which exploit specialized computing and algorithmic resources. Component based programming has gained widespread acceptance in both the industry and academia. The Microsoft COM [22] component frameworks have been fundamental to application interoperability Windows based applications. Now their 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) [27] and Java Beans [26, 6] and EJB [23] have been popular component standards for Java based applications.

Component based models hold great promise to serve as an effective programming model for the Grid. The end user can be provided a rich palette of tools to program by component assembly, rather than by component development, and the lower level details can be handled by the model developers. Our research with XCAT 2.0 (called XCAT herewith) provides an implementation and design of one such model. XCAT addresses Grid requirements by providing a component based programming model (with a scripting interface) that is supported in C++ and Java and a rich set of services that application programmers can use. Thus, the end user only needs to be concerned with user domain problems.

The current trend in Grid middleware is to adopt the emerging web services model. Each Grid service is defined by a Web Service Description Language [2] document and accessed through the protocol mentioned as the binding for the service. This approach is being actively pursued by the Global Grid Forum [12] through the Open Grid Services Architecture (OGSA) project.

The CCA [3] project is an initiative by DOE laboratories and universities to develop a common architecture for building large scale scientific applications from well tested software components. The primary emphasis of CCA has been on building applications and components for massively parallel supercomputers, but there is nothing in the CCA semantics that prohibit it from working on the Grid. Though originally intended to provide a source-level platform, the CCA is now moving to defining a binary-level platform. This will allow a compiled component to run on any conforming implementation.

The central idea in CCA is to build applications by composition. The way two CCA components are composed is by connecting together their ”ports”. Provides ports represent functionality a component provides to other components. 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.

In our previous work [4] we presented an implementation of the CCA specification. It was primarily built as a research vehicle to test the viability of the CCA specification for distributed computing. The system was built using HPC++ [11] and NexusRMI [5] as the underlying communication medium. The binary format of the communication substrate did not lend itself to converting the components as web services. We redesigned and implemented the second version (now called XCAT 2.0) with SOAP [7] as the communication protocol. XCAT focuses on leveraging the advantages of both the component and web services world. It implements all the layers of the Web Service stack. Since the OGSA [1] specification is also based on Web Services, we are interested in exploring the design changes that need to be made so that XCAT components can be OGSA compliant.

The XCAT system is an implementation of the CCA specification. It has been implemented in both C++ and Java and provides seamless interoperability between components written in these two languages.
In this paper we present the following:

1. The design of the different layers of the Web Service protocol stack in XCAT.
2. Description of the various XCAT services.
3. An XCAT based programming model for the Grid.
4. A discussion on XCAT with reference to the OGSA model.

2 XCAT: A CCA based Component Model

The XCAT system is an implementation of the CCA specification. It has been implemented in both C++ and Java and provides seamless interoperability between components written in these two languages. In the following subsection we provide an introduction to CCA concepts and provide a brief comparison with the Corba component model.

2.1 CCA

The CCA [3] is an initiative by DOE laboratories and universities to develop a common architecture for scientific components. It aims to factor out the common functionality of current tools such as CUMULVS [28], PAWS [21], and PARDIS [16] into a standard platform for developing high-performance components. Though originally intended to provide a source-level platform, the CCA is now moving to defining a binary-level platform. This will allow a compiled component to run on any conforming implementation.

CCA components can be composed by connecting their ports. Provides ports represent functionality a component provides to other components. Uses ports represent functionality a component may need. Uses ports are essentially bind-able 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. Each port is identified by name. Figure 1 shows an example of a connection between two components with compatible port types.

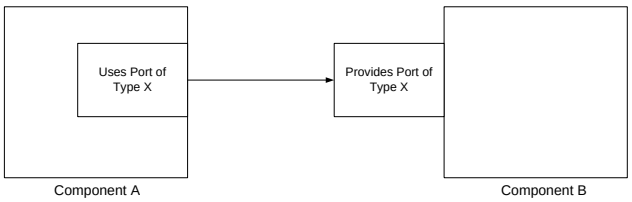

Figure 1. Example of a component connection using CCA. A uses port of type X can be connected to a provides port of the same type.

A CCA component is realized within a run-time system termed a framework. The component interacts with the framework through the Services object. Through this object, the component manages its ports. Ports may be added and removed. The Services object is also used to obtain a connected uses port from the framework.

2.1.1 CCA and Corba CCM Model

The CCA can be compared CORBA Component Model (CCM). Like the CCM, the CCA 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 also be used to build problem-solving environments (PSEs), in which the end-user directly manipulates component connections to solve the particular problem at hand.

Another distinction between the CCM and the CCA is that the CCM is a component layer built on top of a distributed object model. Though this gives some flexibility, and allows the CCM to leverage much of the work of the CORBA object model, it also results in a component model that is somewhat more complex than necessary, especially for scientific applications. The CCA is also moving towards incorporating the idea of a distributed object, but is not defining components in terms of the object model. Rather it is defining distributed objects at an equal level to components.

Finally, the CCA considers parallel applications a top priority. Thus it is working hard to define a specification for collective ports. These ports logically represent one connection, but connecting them may result in multiple network connections being made between components that are composed of a set of parallel operating system constructs, such as processes or threads. Such connections result in higher I/O performance.

3 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. The use of XML messaging systems for interacting with web services is now widely considered as the de facto standard. Web services represent a shift in paradigm from a human-centric to an application-centric web. This does not mean that humans are out of the loop, but just that the interaction between application servers and web browsers can now be automated to a much greater extent.

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

Figure 2. Different layers of the Web Service stack and the example technologies for each layer.

The various protocols composing a Web Service are commonly divided into a five-layer stack as shown in Figure 2. This stack is evolving with various groups working on defining the standards.

1. **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) [33] specification.
2. **Description**: Description of a Web Service includes the available interface, network, transport and packaging protocol that it supports. The Web Service Description Language (WSDL) [2] is a widely accepted standard for this purpose. Resource Description Framework (RDF) [36] is another specification that can be used, though it is less popular than WSDL.
3. **Messaging**: This layer represents the process of marshaling and unmarshaling of application data so that it

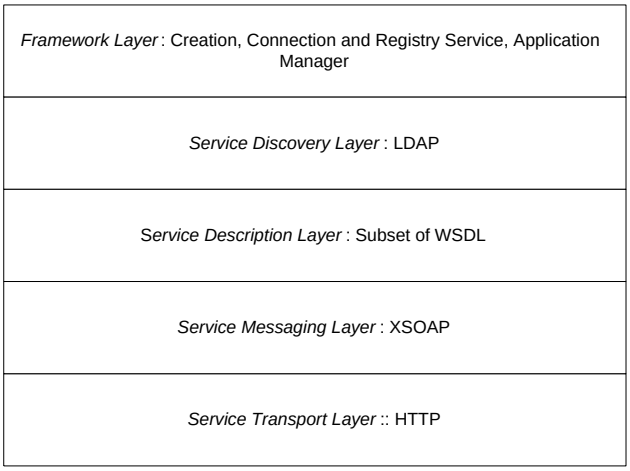

Figure 3. Each layer of the Web Services stack has a corresponding layer in each XCAT component.

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.

4. **Transport**: The transport layer is used to refer to the technology that is used to transfer messages between applications. The choices for this layer include HTTP, SMTP, FTP and BEEP [14].

5. **Framework**: The framework layer provides hooks to other Web Service layers so that applications can use them to build distributed systems. Examples of such frameworks include Microsoft's .NET and Sun Open Net Environment (ONE) [24].

3.1 XCAT and Web Services

Interoperability amongst different implementations of web services is a key concern. Towards this goal, the research community is working towards defining common wire formats (XML), protocols (HTTP and SOAP-RPC) and metadata description (WSDL). With XCAT, we have worked towards incorporating these standards in our implementation and can expose each component as a Web Service. Figure 3 shows how each layer of the web services stack has a corresponding layer in XCAT.

1. **Framework**: XCAT and the CCA provide the realization of the web services framework layer. We describe this in additional detail in Section 3.1.1.

2. **Discovery**: Web services need discovery mechanisms similar to introspection in programming languages. These mechanisms allow clients to examine web services (XCAT components) and discover their properties. XCAT uses an LDAP based registry service (see section 4.3) for the purpose of registering and discovering component instances. We plan to use an UDDI implementation in the near future.

3. **Description**: The interfaces to XCAT components (called ports in the CCA world) are described using XML documents conforming to a Schema. These documents are even 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. Every provides port in the XCAT implementation is a Web Service with one portType. The Web Service is described by a schema that has a subset of the features in WSDL. We are currently in the process of moving to full fledged WSDL for this purpose.

4. **Messaging**: XCAT uses the XSOAP [30] communication system for messaging. It 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 [17] to XCAT. This will give us the option of using a multitude of communication libraries that include SOAP, JMS [31] and binary protocols.

5. **Transport**: Even though the SOAP protocol does not mandate the use of a specific transport protocol, HTTP is the most widely used. Thus, XSOAP also uses the HTTP protocol for transferring messages between applications.

3.1.1 XCAT Framework

1. Framework Implementation: Every remote method call is intercepted by the XCAT-Java framework before it invokes a method on the provides port. This design allows for a security service to be interposed

between the provides port and the XCAT framework. This security service can inspect the call and allow it to go through if the security requirements have been met. The framework makes extensive use of dynamic proxies for every call. This obviates the need for generating glue code for every port type.

2. ComponentID: The ComponentID represents a handle to the component that can be shipped to different locations. XCAT uses the remote reference mechanisms provided by XSOAP to represent an ComponentID. This handle can be *stringified* and stored in registries. It can then be retrieved by interested parties and used to invoke methods on the component. The ComponentID in this serialized form is represented as 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 the course of 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 mapped to language specific exceptions before handing it to the initiating component.
4. XCAT Services: The CCA specification is a light weight specification for software components. It specifies the required behavior for components but does not specify how the components are discovered, created or connected. We have adopted a services based approach for the above, which can easily be modified to conform to the OGSA specifications. The various parts of an XCAT process are shown in Figure 4. The various XCAT Services are described in Section 4.

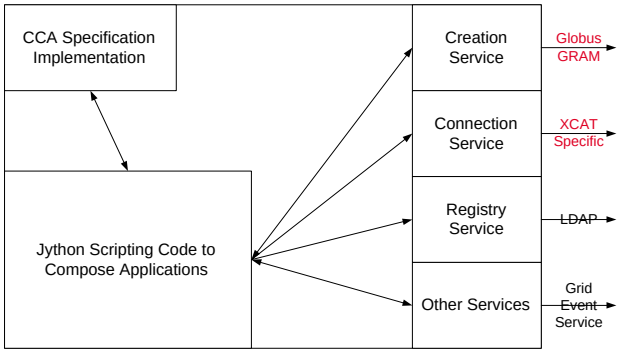

Figure 4. Example of a typical XCAT process that contains user code accessing different services and the implementation of the CCA specification. Each of the services is an interface to a specific implementation.

4 XCAT Services

4.1 Creation Service

The Creation Service is a implementation-specific component that allows a component to instantiate other components. This service completely encapsulates the component instantiation mechanism thus shielding the component developer from the low-level implementation-specific details of the instantiation mechanism. The service exports a provides port (called CreationService (Appendix B) with the following functionality:

1. Create instances of components: The `createInstance` method accepts an `Environment Object` and a timeout value as parameters. The environment object encapsulates the instantiation specific information such as executable location, host machine to execute on, the mechanism to be used for instantiation, command line parameters, process environment variables such stdin, stdout and any other information that the user deems as necessary. It is maintained as a set of (name, value) pairs. Upon successful instantiation of the component, the creation service returns a ComponentID that serves as as handle for the new component. The timeout parameter specifies the time that the creation service waits for the component to be created, after which it throws an exception.
2. Delete instances of components : The `deleteInstance` method destroys the component specified by a given ComponentID.

A new component can be created in the same address space as the creating component or it may be instantiated in a different physical address space on a different host. The instantiation mechanisms available for XCAT are :

1. **GRAM**: This uses the Grid Resource Allocation Management (GRAM) [18] protocol for component creation. The GRAM protocol is provided as a part of the Globus toolkit [20], and we use the Java CoG kit [19] client implementation for this purpose.
2. **(R/S)SH**: Components can be created using the RSH and SSH protocols in which case the components will be launched inside a shell after logging into the target machine using (R/S)SH. This protocol is typically used during the prototyping and debugging stage.
3. **exec**: Components are launched onto the local-host using a *process exec* mechanism. Here the component

runs in a separate address space from the launcher, but on the same host.

4. **proc**: Components that use this mechanism are launched in the same process as the creating component. Future versions of XCAT will provide collocation optimization for calls between components in a single address space.

4.2 Connection Service

The Connection Service (Appendix C) is a framework-specific mechanism by which instantiated components establish communication links with one another via their typed ports. All the methods use the ComponentID to uniquely identify a component. By providing an external mechanism for connecting ports, the port types and descriptions themselves can remain free of any connection semantics. The following is a list of the functionality provided by the service:

1. **Connect and Disconnect**: A component can use this service to connect or disconnect its own ports to another component. It can also connect two other components for which it has the references.
2. **Export**: The Connection Service can also be used to "export" ports of components for which a ComponentID is known, i.e., component A can provide *provides ports* of component B as if they belonged to A. To other components the provides port will appear to belong to A, not to B. Moreover, if a component does not wish to provide one of its ports to the entire world, it can provide it to other components specifically on a "need to know" basis.
3. **Provides as Uses**: The `provideTo()` method provides a *provides port* belonging to the same component as a *uses port* belonging to another component.

4.3 Naming/Registry Service

The Naming or Registry Service (Appendix D) allows components to register references to themselves, which can be retrieved later. The Naming Service is currently based on the Java RMI Registry API. The current interface definition for the Naming Service provides the following features:

1. **Bind and Rebind**: Can be used to bind remote references of components with the registry.
2. **List and Lookup**: This allows users to browse the current bindings in the registry.
3. **Unbind**: To remove an existing binding.

Currently, we are working on incorporating authentication and authorization for retrieving the references, and modifying the implementation to conform to the JNDI API [25], so that we can use hierarchical naming schemes, for greater scalability.

4.4 Application Manager Service

The scriptable application manager is a generic XCAT component, which can act as a controller for the actual applications. The actual functionality of the manager is defined by the script it loads and executes inside a Jython interpreter, and this is the way it differs from other existing XCAT components. This leads to the ability to change the behavior of the manager at run time, and the ability to manage various applications with minimal compilation. The manager also has the capability of sprouting predefined CCA ports at runtime, so that it can communicate with other CCA components, and use the services provided by them.

The application manager (Appendix E) contains a Jython interpreter into which the remote script gets downloaded. The manager exports a CCA provides port called the ScriptPort, which has methods that are required to manage the applications. In particular, it has a method `runScript()`, which downloads the script into the interpreter, and begins execution. This returns an integer to the callee, identifying the script execution. Using this identifier, the script in execution can be killed using the `killScript()` method. The `runScript()` method can be blocking or non-blocking.
The application managers also have a variable number of MesgPorts, through which they can send messages to other components which provide such MesgPorts. Each application manager has 'n' number of UsesMesgPorts, if they need to send messages to 'n' other components. For receiving messages, each application manager has a ProvidesMesgPort, which receives the messages and stores it in a buffer for the scripts to process. The scripts can retrieve these messages from the application managers, by invoking the `getMessage()` method on the manager. The scripts can send messages, using the `sendMessage()` method of the manager.

The various administrative functions for the application managers are handled by the AppManPort and the ControlPort, that each application manager provides. The AppManPort provides methods to dynamically add provides and uses ports using the addProvidesPort() and addUsesPort() methods respectively. This capability of the application manager enables the scripts running inside it to communicate with any other XCAT component.

The createJavaComponent() and createCppComponent() methods can be used to create Java and C++ components respectively, on the machine the application manager is running on. Finally, the setNumMesgListeners() method can be used to set the total number of uses MesgPorts that the application manager should have. The ControlPort provides methods start() and kill(), which have obvious meanings.

The following are the salient features of the application managers.

- They have a variable of ports that can be used to sent messages to interested components.
- They allow ports to be dynamically added.
- Allow the scripts running within the component to communicate with other XCAT components.
- Provide mechanism for creating other components.

The application managers have been used successfully to manage a variety of applications, e.g. the first generation NCSA chemical engineering applications [9], the IU Xports application [9], the NCSA Weather Research and Forecasting (WRF) applications, and the next generation NCSA chemical engineering application, which is described in section **??**.

5 Programming Model

To use a component architecture effectively, it is important to be able to describe the various components that constitute an application along with its interconnections. It should be possible to create "metacomponents" which are themselves created by composing a number of components together. XCAT provides a Jython based programming model for the end user to create components and orchestrate computations. The programming model consists of a set of APIs to facilitate user access the entire range of XCAT functionality via Jython scripts. In this section we present the programming model and show an example of its usage. XCAT also provides the conventional programming model of using Java based programs to create and control applications. We describe both the approaches in this section.

5.1 Java Control Programs

The XCAT Services APIs can be used directly by the user to write simple Java control programs. The user can use the Creation Service to create components and get references to running instances. The user can then use the Connection Service interface 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.

Snippets from a sample java control program is shown in Appendix A. The listed code shows how two components, a printer and a generator can be created using the createInstance() method of the Creation Service, The "testObjectUsesPort" of the generator is then connected with the "testObjectProvidesPort" of the printer, using the connect() method of the Connection Service. A uses port of type "UsesControl" is added dynamically using the `registerUsesPort()` method, and it is connected to the same provides port of the generator component. The `start()` method is then invoked on the uses port, to start the execution of the components.

5.2 Jython Scripting

The above method (of using Java control programs) is only suitable when we have a fixed set of components, which are to be launched, and monitored. It is desirable to have a more dynamic mechanism to create and manage components on the fly, without the need for any recompilation. 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. The list of various features provided by the scripting interface include the following:

1. Provide the name of the host on which the component needs to be instantiated.
2. Set a creation mechanism that will be used to launch the component on the Grid.
3. Get a handle to the various XCAT services.
4. Launch a component on the host using the requested mechanism.
5. Connect and disconnect ports of different components.
6. Invoke methods on the ports of various components.

Apart from the above two ways to orchestrate computations (Java control programs and Jython scripts), applications specific GUIs can be easily written and layered on top of the services provided by XCAT. We are also working on a workflow description for computations so that the end user does not need to write any Java or Jython code, but rather interact with the system, using a simple client, e.g. a web browser. We are investigating techniques like DAGMan [34] and WSFL [13] for the same.

6 XCAT and OGSA

The Open Grid Services Architecture (OGSA) specification [1] represents an effort to develop a model that meets

the challenges of integrating services across a distributed and heterogeneous environment. It builds on top of web services with a set of conventions (interfaces and behaviors) that define the interaction of clients with OGSA services. These conventions include a set of useful standard interfaces to discover service metadata, control service lifecycle (using GridService portType) and to create new service instances (using Factory portType). In this section we discuss some OGSA features with reference to XCAT.

1. **Port Type**: In XCAT the port type is represented as a unique string that refers to the name of the XML schema document that describes it. The OGSA port type is identified by a QName, which is also a string.
2. **GridService Port Type**: This port type must be supported by every OGSA complaint service. This port type contains methods used for service lifetime management and introspection. The XCAT framework creates a few services by default for every component, creation and connection services for example. It will be easy to add the requirement for a GridService port type as a default service provided by every XCAT component.
3. **OGSA Services**: The OGSA services are analogous to the provides ports in the XCAT world. XCAT based applications are composed by connecting compatible uses and provides ports. Web services can be viewed as unnamed provides ports and specialized factories can be used to compose applications consisting of XCAT components and OGSA services.
4. **Introspection**: Web services need discovery mechanisms similar to introspection in programming languages. These mechanisms allow clients to examine web services to discover their properties. In the OGSA world this data is called service metadata. It can be retrieved by invoking the `findServiceData()` method on the GridService port type. XCAT provides an API to retrieve meta information on ports that includes names, types and properties.

7 Sample Applications

The XCAT Science Portal [9] uses the XCAT implementation as the underlying model for launching distributed applications on the grid. Some applications that use the XCAT system are the IU Xports [9], the NCSA Weather Research and Forecasting (WRF), GRAPPA [15], the Collision Risk Assessment (CRASS) system [8], and the Linear Systems Analysis (LSA) project [8].

The following subsection describes the NCSA Chemical Engineering application, which describes a typical scenario in which XCAT can be used.

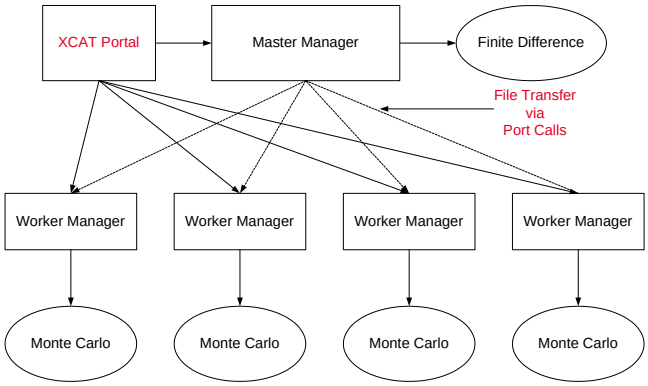

Figure 5. Chemical Engineering Code Linkage

7.1 The NCSA Chemical Engineering Application

The chemical engineers at NCSA aim to link a finite difference electrical resistance code to multiple Monte Carlo electro-deposition simulations, running on resources over the grid. The number of Monte Carlo simulations may be dynamically changed at runtime, i.e. there is a need for the codes to be dynamically composed at runtime. These codes also need to exchange data at every iteration. The linkage system needs to be designed such that the application writer has to make minimal changes to the application code. In other words, it should be possible for the chemical engineers to write the finite difference code, and the Monte Carlo code, without any knowledge of the fact that they will be executed on the grid.

The XCAT system lends itself very well to the needs of this application because XCAT components can be launched on the grid, and can be composed together dynamically at runtime. We decided to use the concept of the application manager service, as illustrated in Section 4.4, to wrap up and manage them. We used application managers, which are scriptable, and also specialized "Master/Worker" application managers for our purposes.

As seen in Figure 5, the master application manager manages the finite difference code, while there is a worker application manager for each of the Monte Carlo simulations. The actual scientific codes are not modified, but they are assumed to exhibit behavior that is known to the managers, e.g. they write out files after every iteration, with data that has to be transferred between themselves, and read data from predefined filenames for each iteration. The master manager has a ProvidesMasterPort, which is capable of accepting data sent to it by a worker manager (which holds a corresponding uses port) after the end

of every iteration. On receiving data via the port call, the master manager writes it out to a predefined file for the executable to consume. Each worker manager has a ProvidesWorkerPort, which can accept data sent to it by the master manager at the end of each iteration. The worker manager, as expected, writes the data out to a file, that the executable expects.

Furthermore, these managers monitor the state of the execution, and can send events about the status to a well known event channel. The overall control over the application managers is via a Jython script, which launches these managers (and hence, the applications) on the grid resources, using the XCAT creation service, and connects the ports together, to bootstrap the simulation. In summary, the chemical engineers were able to successfully achieve the linkages dynamically, and run the simulations on the grid, without making changes to the scientific codes themselves to make them grid-enabled.

8 Future Work

XCAT provides a default set of services to each component. We plan to add authentication, authorization, factory, and scheduling services. Furthermore, we will make XCAT an OGSA compliant system.
XCAT can also be used to connect components that run in the same address space. In such cases it is important to provide collocation optimization for efficient communication within the process. We plan to add this feature to the next version of XCAT.

With the availability of WSDL parsers, it is now possible to represent remote references to ports and components using WSDL documents. WSDL will also provide a way of representing various protocols that can be used for communication between components. We will use the Proteus library [17] to achieve multiprotocol communication between components.

9 Conclusions

We have presented XCAT as a component based, language-interoperable and scriptable framework that can serve as an effective programming model for the Grid. In XCAT, ports can be dynamically added and deleted, allowing component connections and application configurations to be changed at run-time. Thus, complex scientific applications can be made up of simpler components, which can be composed to achieve the desired result. Every XCAT component has access to a set of services by default that include the creation, connection and registry services. The creation service offers a multitude of mechanisms for component creation, which include GRAM, SSH and local instantiations. The connection service allows a component to export ports of a sub-component as its own, apart from its basic functionality which is to connect ports between components. XCAT provides a Jython based scripting interface that facilitates application development. We have also presented a typical application for which the XCAT system is applicable, demonstrating the use of the concept of application managers. We have presented a case for XCAT as the model for handling transient and stateful Grid web services, which involve a high degree of interaction between each other.

References

[1] Argonne National Lab. The Open Grid Services Architecture, visited 03-03-02. http://www.globus.org/ogsa/.

[2] Ariba, IBM, Microsoft. Web Services Description Language (WSDL) Version 1.1, visited 02-01-02. http://www.w3.org/TR/wsdl.

[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] Randall Bramley, Kenneth Chiu, Shridhar Diwan, Dennis Gannon, Madhusudhan Govindaraju, Nirmal Mukhi, Benjamin Temko, and Madhuri Yechuri. A component based services architecture for building distributed applications. In *Proceedings of Ninth IEEE International Symposium on High Performance Distributed Computing Conference, Pittsburgh*, August 1-4 2000.

[5] Fabian Breg, Shridhar Diwan, Juan Villacis, Jayashree Balasubramanian, Esra Akman, and Dennis Gannon. Java RMI performance and object model interoperability: Experiments with Java/HPC++. *Concurrency and Experience*, 1998. Presented at 1998 ACM Workshop on Java for High-Performance Network Computing.

[6] Robert Englander. *Developing Java Beans*. O'Reilly, 1997.

[7] D. Box et al. Simple Object Access Protocol 1.1. Technical report, W3C, 2000. http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

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

[9] S. Krishnan et al. The XCAT Science Portal. In *Proceedings of SuperComputing 2001, Denver, CO, 2001*, November 2000.

[10] Ian Foster and Carl Kesselman. *The GRID: Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 1998.

[11] 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.

[12] GGF. Global Grid Forum, 06-01-02. http://www.gridforum.org/.

[13] IBM. Web Services Flow Language (WSFL) Version 1.0, visited 02-01-02. www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf.

[14] IETF. The Blocks Extensible Exchange Protocol Core, Visited 05-20-02. http://www.ietf.org/rfc/rfc3080.txt.

[15] Indiana University. Grid Access Portal for Physics Applications, visited 03-03-02. http://iuatlas.physics.indiana.edu/grappa/.

[16] Katarzyna Keahey and Dennis Gannon. PARDIS: A Parallel Approach to CORBA. In *Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing*, August 1997.

[17] Kenneth Chiu and Madhusudhan Govindaraju and Dennis Gannon. The Proteus Multiprotocol Library, 2002.

[18] Argonne National Lab. GRAM, visited 01-03-02. http://www.globus.org/gram.

[19] Argonne National Lab. Java Cog Toolkit, visited 01-03-02. http://www.globus.org/cog.

[20] Argonne National Lab. Globus, visited 1-02-01. http://www.globus.org.

[21] Los Alamos National Lab. PAWS : Parallel Application Workspace, visited 03-03-02. http://www.acl.lanl.gov/paws/.

[22] Microsoft. COM, visited 4-1-2000. http://www.microsoft.com/com.

[23] Sun Microsystems. EJB, visited 02-01-02. http://java.sun.com/products/ejb/index.html.

[24] SUN Microsystems. Sun Open Net Environment, visited 04-15-02. http://wwws.sun.com/software/sunone/.

[25] SUN Microsystems. JNDI, visited 3-7-2002. http://java.sun.com/products/jndi/.

[26] SUN Microsystems. Java Beans, visited 4-15-00. http://java.sun.com/beans/.

[27] OMG. Corba Component Model, visited 1-11-2000. http://www.omg.org/cgi-bin/doc?orbos/97-06-12.

[28] ORNL. CUMULVS: Collaborative User Migration, User Library for Visualization and Steering, visited 03-03-02. http://www.csm.ornl.gov/cs/cumulvs.html.

[29] Rice University. High Performance Fortran (HPF), visited 03-01-02. http://www.crpc.rice.edu/HPFF/home.html.

[30] 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.

[31] Sun Microsystems. Java Messaging Service, 04-20-02. http://java.sun.com/products/jms/.

[32] The MPI Forum. MPI Documents, visited 03-03-02. http://www.mpi-forum.org/docs/docs.html.

[33] UDDI.org. Universal Description Discovery and Integration of Business for the Web, 04-12-02. http://www.uddi.org/specification.html.

[34] University of Wisconsin. DAGMan : Directed Acyclic Graph Manager, visited 03-01-02. http://www.cs.wisc.edu/condor/dagman/.

[35] University of Wisconsin. The Condor Project Homepage, visited 03-01-02. http://www.cs.wisc.edu/condor/.

[36] W3C. Resource Description Framework, 06-01-02. http://www.w3.org/RDF/.

A Snippet of Java code to create and connect components

```
....
// create the components using the Creation Service
ComponentID printerCompID =
 creationService.createInstance(envPrinterComponent);

ComponentID generatorCompID =
 creationService.createInstance(envObjectComponent);

// connect ports using the Connection Service
connectionService.connect(generatorCompID,
                          ‘‘testObjectUsesPort",
                          printerCompID,
                          "testObjectProvidesPort");

// register a uses port to invoke method on
services.registerUsesPort
 (new PortInfoImpl("controlUsesPort", WSDL_DESCRIPTION));

// connect the uses port just registered, to the remote component
connectionService.connect(this.componentID,
                          "controlUsesPort",
                          generatorCompID,
                          "controlProvidesPort");

// get a reference to the uses port
UsesControl control =
 (UsesControl) services.getPort("controlUsesPort");

// start the execution of the generator component
// this will bootstrap the execution of the components
control.start();
```

B Creation Service API

```
public interface CreationService extends Port {
  public ComponentID createInstance(EnvObj env, int timeout)
    throws XCATException;
  public void deleteInstance(ComponentID id)
    throws XCATException;
}
```

C Connection Service API

```
public interface ConnectionService extends Port {
  public void connect(ComponentID idUser,
                      String usesportname,
                      ComponentID idProv,
                      String provportName)
    throws RemoteException;
```

```
  public void disconnect(ComponentID idUser,
                         String usesportname,
                         ComponentID idProv,
                         String provportname, double timeout)
    throws RemoteException;
  public void exportAs(ComponentID idMe,
                       ComponentID idSub,
                       String portname,
                       String newportname)
    throws RemoteException;
  public void provideTo(ComponentID idSub,
                        String usesPortName,
                        ProvidesPort providesPort)
    throws RemoteException;
}
```

D Naming Service API

```
public interface NamingService extends Port {
  public void bind(String name, String ref)
    throws XCATException;
  public void rebind(String name, String ref)
    throws XCATException;
  public String[] list(String name)
    throws XCATException;
  public String lookup(String name)
    throws XCATException;
  public void unbind(String name)
    throws XCATException;
}
```

E Application Manager API

```
public interface ScriptPort_idl extends Port {
  public void setParams(Object[] params)
    throws RemoteException;
  public int runScript(String script)
    throws RemoteException;
  public int runScriptBlocking(String script)
    throws RemoteException;
  public int killScript(Integer id)
    throws RemoteException;
}
```