Grid Web Services and Application Factories

Dennis Gannon, Rachana Ananthakrishnan, Sriram Krishnan, Madhusudhan Govindaraju
Lavanya Ramakrishnan, Aleksander Slominski
Department of Computer Science
Indiana University

Abstract. This paper describes an implementation of a Grid Application Factory Service that is based on a component architecture that utilizes the emerging Web Services standards. The factory service is used by Grid clients to authenticate and authorize a user to configure and launch an instance of a distributed application. This helps us solve the problem of building reliable, scalable Grid applications, by separating the process of deployment and hosting from application execution. The paper also describes how these component-based applications can be made compatible with the Open Grid Service Architecture(OGSA) and how OGSA concepts enhance the usability of the component framework.

1. Introduction.

A Grid can be defined as a layer of networked services that allow users single sign-on access to a distributed collection of compute, data and application resources. The Grid services allow the entire collection to be seen as a seamless information processing system that the user can access from any location. Unfortunately, for application developers, this Grid vision has been a rather elusive goal. The problem is that while there are several good frameworks for Grid architectures (Globus [3] and Legion/Avaki [3,7]), the task of application development and deployment has not become easier. The heterogeneous nature of the underlying resources remains a significant barrier. Scientific applications often require extensive collections of libraries that are installed in different ways on different platforms. Moreover unix-based default user environments vary radically between different users and even between the user's interactive environment and the default environment provided in a batch queue. Consequently, it is almost impossible for one application developer to hand an execution script and an executable to another user and expect the second user to be able to successfully run the program on the same machine, let alone a different machine on the Grid. The problem becomes even more complex when the application is a distributed computation that requires a user to successfully launch a heterogeneous collection of applications on remote resources. Failure is the norm and it can take days, if not weeks to track down all the incorrectly set environment variables and path names.

A different approach, and the one advocated in this paper, is based on the web services model [7,8,9,10], which is quickly gaining attention in industry. The key idea is to isolate the responsibility of deployment and instantiation of a component in a distributed computation from the user of that component. In a web service model, the users are only responsible for accessing running services. The Globus Toolkit provides a service for the remote execution of a job, but it does not attempt to provide a standard hosting environment that will guarantee that the job executes correctly. That task is left to the user. In a web service model, the job execution and lifetime becomes the responsibility of the service provider.

The recently proposed Open Grid Service Architecture (OGSA) [1,2] provides a new framework for thinking about and building Grid applications that are consistent with this service model view of applications. OGSA specifies three things that a web service must have before it qualifies as a Grid Services. First it must be an instance of a service implementation of some service type as described above. Second, it must have a Grid Services Handle (GSH), which is a type of Grid URI for the service instance. The third property that elevates a Grid Service above a garden-variety web service is the fact that each Grid Service instance must implement a port called "GridService", which provides any client access to service metadata and service state information. In the following section of this paper we will describe the role the GridService port can play in a distributed component system.

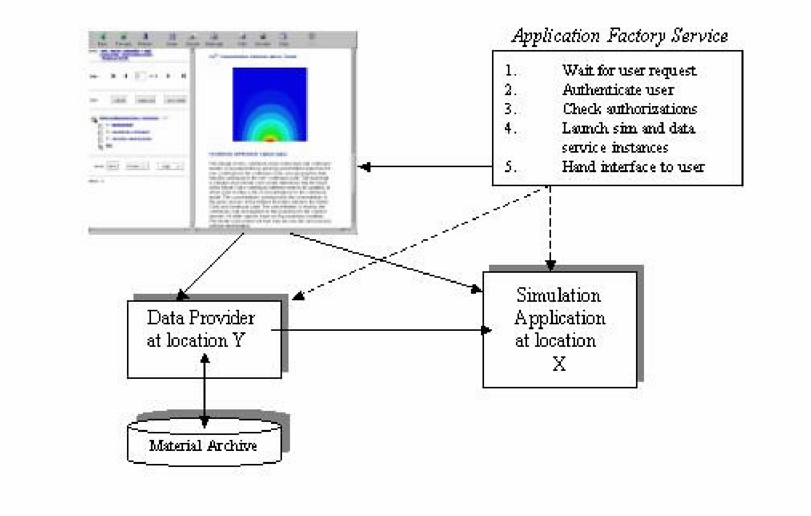
OGSA also provides several other important services and port types. Messaging is handled by the NotificationSource and NotificationSink ports. The intent of this service is to provide a simple publish-subscribe system similar to JMS[17], but based on XML messages. A Registry service allows other services to publish service metadata and to register services. From the perspective of this paper, a very important addition is the OGSA concept of a Factory service, which is used to create instances of other services.

In this paper, we describe an implementation of an Application Factory Service that is designed to create instances of distributed applications that are composed of well-tested and deployed components each executing in a well-understood and predictable hosting environment. In this model both the executing component instances and the composite application are web services. We also describe how some important features of OGSA can be used to simplify client access to the running application from a conventional web portal. We also describe a simple security model for the system that is designed to provide both authentication and simple authorization. We conclude with a discussion of how the factory service can be used to isolate the user from the details of resource selection and management in Grid environments.

1.1 An overview of the Application Factory Service

The concept of a Factory Service is not new. It is an extension of the Factory Design Pattern [13] to the domain of distributed system. A factory service is a secure and stateless persistent service that knows how to create an instance of transient, possibly stateful service. Clients contacts the factory service and supply the needed parameters to instantiate the application instance. It is the job of the service to invoke exactly one instance of the application and return a WSDL document that clients can use to access the application. OGSA has a standard port type for factory services, which has the same goal as the one described here but the details differ in some respects.

To illustrate the basic concept we begin with an example. Suppose a scientist at a location X has a simulation code that is capable of doing some interesting computation provided it is supplied with useful initial and bound conditions. A supplier at another location Y may have a special data archive that describes material properties that define possible boundary or initial conditions for this simulation. For example, these may be aero-dynamic boundary conditions such as fluid temperature and viscosity used in a simulation of turbulence around a solid body, or process parameters used in a simulation of a semi-conductor manufacturing facility. Suppose the supplier at Y would like to provide users at other locations with access to the application that uses the data archive at Y to drive the simulation at X. Furthermore suppose that the scientist at location X is willing to allow others to execute his application on his resources, provided he authorizes them to do so.



Application Factory Service
1. Wait for user request
2. Authenticate user
3. Check authorizations
4. Launch sim and data service instances
5. Hand interface to user
Data Provider at location Y
Simulation Application at location X
Material Archive

Figure 1. High-level view of user/application factory service. User contacts the persistent factory service from a web interface. Factory service handles authentication and authorization and then creates an instance of the distributed application. A handle the distributed application is returned to the user.

To understand the requirements for building such a grid simulation service we can follow a simple use-case scenario.

- The user would contact the factory service through a secure web portal or a direct secure connection from a factory service client. In any case, the factory service must be able to authenticate the identity of the user.
- Once the identity of the user has been established the factory service must verify that the user is authorized to run the simulation service. This authorization may be as simple as checking an internal access control list, or it may involve consulting an external authorization service.
- If the authorization check is successful the factory service can allow the user to communicate any basic configuration requirements back to the factory service. These configuration requirements may include some basic information such as

estimates of the size of the computation or simulation performance requirements that may affect the way the factory service selects resources on which the simulation will run.

- The factory service then starts a process, which creates running instances of a data provider component at Y and a simulation component at X that can communicate with each other. This task of activating the distributed application may require the factory service to consult resource selectors and workload managers to optimize the use of compute and data resources. For Grid systems, there is an important question here: under whose ownership are these two remote services run? In a classic grid model, we would require the end user to have an account on both the X and Y resources. In this model the factory service would now need to obtain a proxy certificate from the user to start the computations on the user's behalf. However, this delegation is unnecessary if the resource providers trust the factory service and allow the computations to be executed under the service owner's identity. The end users need not have an account on the remote resources and this is a much more practical service oriented model.
- Access to this distributed application is then passed from the factory service back to the client. The easiest way to do this is to view the entire distributed application instance as a transient, stateful web service that belongs to the client.
- The factory service is now ready to interact with another client.

In the sections that follow we describe the basic technology used to build such a factory service. The core infrastructure used in this work is based on XCAT [14,15], which is a Grid-level implementation of the Common Component Architecture [16] developed for the U.S. Department of Energy. XCAT can be thought of as a tool to build distributed application oriented web-services. We also describe how OGSA related concepts can be used to build active control interfaces to these distributed applications.

2. XCAT and Web Services

In this section we describe the component model used by XCAT and discuss its relation to the standard web service model and OGSA. XCAT components are software modules that provide part of a distributed application's functionality in a manner similar to that of a class library in a conventional application. A running instance of an XCAT component is a web service that has two types of ports. One type of port, called a **provides-port**, is essentially identical to a normal web service port. A provides-port is a service provided by the component. The second type of port is called a **uses-port**. These are ports that are "outgoing only" and they are used by one component to invoke the services of another, or as will be described later, to send a message to any waiting listeners. Within the CCA model, as illustrated in Figure 2, a uses-port on one component may be **connected** to a provides-port of another component if they have the same port interface type.

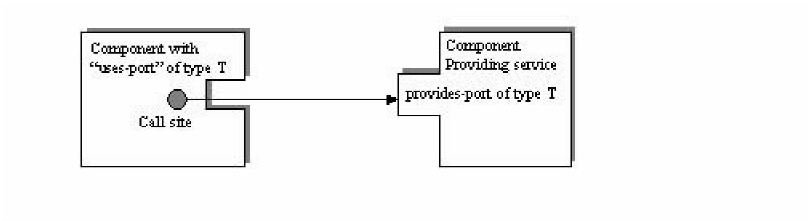


Figure 2. CCA Composition model. A uses-port, which represents a proxy for an invocation of a remote service may by bound at runtime to any provides-port of the same type on another component.

Furthermore this connection is dynamic and it can be modified at run-time. The provides-ports of an XCAT component can be described by the Web Service Description Language (WSDL) and hence can be accessed by any web service client that understands that port type. (A library to generate WSDL describing any remote reference is included as a part of XSOAP[18], which is an implementation of Java Remote Method Protocol (JRMP) in both C++ and Java with SOAP as the communication protocol. Since, in XCAT a provides-port is a remote reference, the XSOAP library can be used to obtain WSDL for any provides-port. Further, a WSDL describing the entire component, which includes the WSDL for each provides port, can be generated using this library.) The CCA/XCAT framework allows

- any component to create instances of other components on remote resources where it is authorized to do so, (In XCAT this is accomplished using Grid services such as Globus)
- any component to connect together the uses/provides ports of other component instances (when it is authorized to do so), and
- a component to create new uses and provides ports as needed dynamically.

These dynamic connection capabilities make it possible to build applications in ways not possible with the standard web services model. To illustrate this, we compare the construction of a distributed application using the CCA/XCAT framework and web services using the Web Services Flow Language (WSFL) [10] which is one of the leading approaches to combining web services into composite applications.

Typically a dynamically created and connected set of component instances represents a distributed application that has been invoked on behalf of some user or group of users. It is stateful and, typically, not persistent. For example, suppose an engineering design team wishes to build a distributed application that starts with a database query which provides initialization information to a data analysis application which frequently needs information found in a third party information service. An application coordinator component (which will be described later in greater detail) can be written that instantiates an instance of a database query component, a specialized legacy program driver component and a component that consults a third party data service, all connected as shown in Figure 3. Suppose the operation of this data analysis application is as follows. The database query component provides a web-service interface to users and when invoked by a user, it consults the database and contacts the analysis component. The analysis component, when receiving this information, interacts periodically with the data service and eventually returns a result to the database component, which returns it to the user.

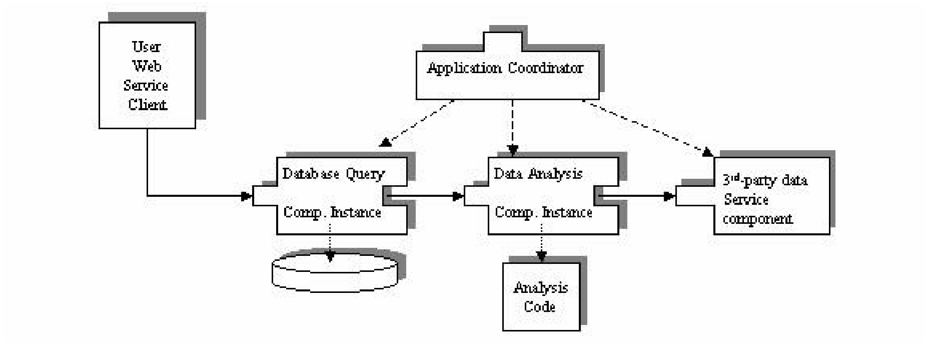

User Web Service Client
Application Coordinator
Database Query Comp. Instance
Data Analysis Comp. Instance
3rd-party data Service component
Analysis Code

Figure 3. A data analysis application. An Application Coordinator instantiates three components: a database query component, a data analysis component which manages a legacy application and a third party data service (which may be a conventional web service).

This entire ensemble of connected component instances represents a distributed, transient service that may be accessed by one user or group of users and may exists for only the duration of a few transactions.

In the case above, the use of the application controller component to instantiate and connect together a chain of other components is analogous to a workflow engine executing a WSFL script on a set of conventional web services. As shown in Figure 4, the primary advantage of the CCA component model is that the WSFL engine must intermediate at each step of application sequence and relay the messages from one service to the next.

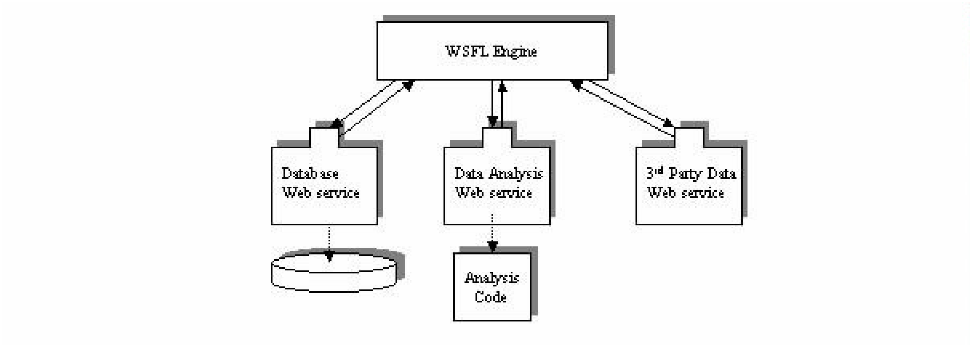


Figure 4. Standard Web Service linking model using a Web Service Flow Language document to drive a WSFL Engine.

If the data traffic between the services is heavy it is probably not best to require it to go through a central flow engine. Furthermore, if logic that describes the interaction between the data analysis component and the third party data service is complex and depends upon the application behavior, then putting it in the high level workflow may not work. This is an important distinction between application dependent flow between components and service mediation at the level of workflow.

In our current implementation each application is described by three documents.

- The **Static Application Information** is an XML document that describes the list of components used in the computation, how they are to be connected and the ports of the ensemble that are to be exported as application ports.
- The **Dynamic Application Information** is another XML document that describes the bindings of component instances to specific hosts and other initialization data.
- The **Component Static Information** is an XML document that contains basic type information about the component and all the details of its execution environment for each host on which it has been deployed. This is the information that is necessary for the application coordinator component to create a running instance of the component. The usual way to obtain this document is through a call to a simple directory service, called the **Component Browser**, which allows a user to browse components by type name or to search for components by other attributes such as the port types they support.

To illustrate the way the Static and Dynamic Application Information is used, consider the small example of the data analysis application above. The static application information, shown below, just lists the components and the connections between their ports. Each component is identified both by type and by the component browser from which its static component information is found.

```
<application appName="Data Analysis Application">
   <applicationCoordinator>
       <component name="application coordinator">
          <compID>AppCoordinator</compID>
          <directoryService>uri-for-comp-browser</directoryService>
       </component>
   </applicationCoordinator>
   <applicationComponents>
       <component name="Database Query">
           <compID>DBQuery</compID>
          <directoryService>uri-for-comp-browser</directoryService>
       </component>
       <component name="Data Analysis">
          <compID>DataAnalysis</compID>
          <directoryService>uri-for-comp-browser</directoryService>
       </component>
       <component name="Joe's third party data source">
          <compID>GenericDataService</compID>
          <directoryService>uri-for-comp-browser</directoryService>
      </component>
  </applicationComponents>
   <applicationConnections>
       <connection>
           <portDescription>
               <portName>DataOut</portName>
               <compName>Database Query</compName>
           </portDescription>
            <portDescription>
               <portName>DataIn</portName>
               <compName>Data Analysis</compName>
           </portDescription>
      </connection>
       <connection>
           <portDescription>
               <portName>Fetch Data</portName>
               <compName>Data Analysis</compName>
           </portDescription>
            <portDescription>
               <portName>Data Request</portName>
               <compName>Joe's third party data source</compName>
           </portDescription>
      </connection>
   </applicationConnections>
</application>
```

The dynamic information document simply binds components to hosts based on availability of resources and authorization of user. For the example described above, it may look like this:

```
<applicationInstance name="Data Analysis Application">
   <appCoordinatorHost>application coordinator</appCoordinatorHost>
   <compInfo>
      <componentName>Data Query</componentName>
      <hostName>rainier.extreme.indiana.edu</hostName>
   </compInfo>
   <compInfo>
      <componentName>Data Analysis</componentName>
      <hostName>modi4.csrd.uiuc.edu</hostName>
   </compInfo>
   <compInfo>
      <componentName>Joe's third party data source</componentName>
      <hostName>joes_data.com</hostName>
   </compInfo>
</applicationInstance>
```

An extension to this dynamic information application instance document provides a way to supply any initial configuration parameters that are essential for the operation of the component.

These documents are used by the application coordinator to instantiate the individual components. The way in which these documents are created and passed to the coordinator is described in detail in the next two sections.

2.1 The OGSA Grid Services Port and standard CCA ports

To understand how the CCA/XCAT component framework relates to the Open Grid Service Architecture, one must look at the required features of an OGSA service. In this paper we focus on one aspect of this question. The Open Grid Services Architecture requires that each service that is a fully qualified OGSA service must have a port that implements the GridService port. This port implements four operations. Three of these operations deal with service lifetime and one, **findServiceData,** is used to access service metadata and state information. The message associated with the findServiceData operation is a simple query to search for serviceData objects, which take the form shown below.

```
<gsdl:serviceData name="nmtoken"? globalName="qname"? type="qname"
      goodFrom="xsd:dateTime"? goodUntil="xsd:dateTime"?
      availableUntil="xsd:dateTime"?>
   <-- content element --> *
</gsdl:serviceData>
```

The **type** of a serviceData element is the XML schema name for the content of the element. Hence almost any type of data may be described here. OGSA defines about ten different required serviceData types and we agree with most of them. There are two standard default serviceData search queries: finding a serviceData element by name and finding all serviceData elements that are of a particular type. This mechanism provides a very powerful and uniform way to allow for service reflection/introspection. Though not implemented as a standard port in the current XCAT implementation, it will be added in the next release. The XCAT serviceData elements will contain the static component information record associated with its deployment. An important standard XCAT component serviceData element is a description of each port that the component supports. This includes the port name, the WSDL port type, whether it is a provides-port or a uses-port and if it is a uses port whether it is currently connected to a provides-port or not. Often component instances publish event streams that are typed XML messages. An important serviceData element contains a list of all the event types and the handle for the persistent event channel that stores the published events.

Many application components also provide a custom control port that allows the user to directly interact with the running instance of the component. In this case a special ControlDocument serviceData element can be used to supply a user with a graphical user interface to the control port. This user interface can be either a downloadable applet-like program or a set of web pages and execution scripts that can be dynamically loaded into a portal server such as the XCAT science portal [14]. As illustrated in Figure 5, this allows the user control of the remote component from a desktop web browser.

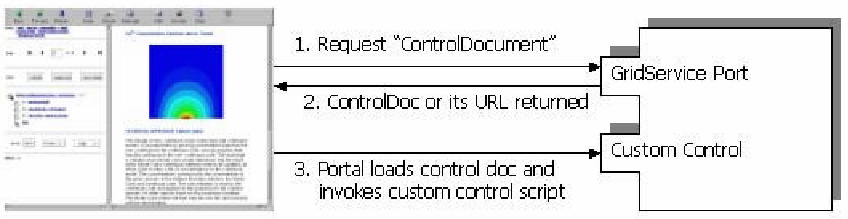

1. Request "ControlDocument"
2. ControlDoc or its URL returned
3. Portal loads control doc and invokes custom control script
GridService Port
Custom Control

Figure 5. Portal interaction with GridService port.

Within the web services community there is an effort called Web Services for Remote Portals (WSRP), which is attempting to address this problem [12]. The goal of this effort is to provide a generic portlet, which runs in a portal server and acts as a proxy for a service-specific remote portlet. As this effort matures we will incorporate this standard into our model for component interaction.

Each XCAT component has another standard provides-port called the **Go** port. The life cycle of a CCA component is controlled by its Go. There are three standard operations in this port

- **sendParameter,** which is used to set initialization parameters of the component. The argument is an array of type Object, which is then cast to the appropriate specific types by the component instance. A standard serviceData element for each component is "parameters", which provides a list of tuples of the form (name, type, default, current) for each component parameter.
- **start,** which causes the component to start running. In the CCA model, a component is first instantiated. At this point only the GridService, start and the sendParameter port are considered operational. Once parameters are set by the sendParameter method (or defaults are used) and the start method has been invoked then the other ports will start accepting calls. These rules are only used for stateful components or stateless components that require some initial, constant state.
- **kill,** which shuts down a component. If the start method registered the component with an information service, this method will also un-register the instance. In some cases, kill will also disconnect child components and kill them.

(It should be noted that in the CCA/XCAT framework we have service/component lifetime management in the Go port, while in OGSA it is part of the GridService port. Also, the OGSA lifetime management model is different.)

2.1.2 Application Coordinator.

Each XCAT component has one or more additional ports that are specific to its function. The Application Coordinator, discussed in the multi-component example above, has the following port:

ACCreationProvidesPort

This port is a provides-port that provides the functionality to create applications by instantiating the individual components that make up the application, connecting the uses and provides port of the different components and generating WSDL describing the application. This functionality is captured in the following two methods:

- **createApplication**, which accepts the static and dynamic XML describing the application and parameters. The XML strings are parsed to obtain installation information and host name for each component. The components are launched using the XCAT creation service and the appropriate execution environment values. Further, information about the connections between the different components is extracted from the static XML and using the XCAT connection service the connections are established. The static XML also has information about the various ports of the components that need to be exposed to the clients. These ports, that belong to different components, are exported as ports of the Application Coordinator. In the next version the static XML will also contain a list of bootstrap methods that need to be invoked on the individual components to start up the application. Once the application has been successfully instantiated, bootstrap methods, if any are invoked.

- **generateWSDL**, which returns the WSDL describing the application as a whole. The ports of the Application Coordinator component and the exported ports are included as a part of this WSDL.

Another port, the **ACMonitorProvidesPort**, has methods like "killApplication" and "pingApplication" for monitoring the application. The "killApplication" method parses the static XML to retrieve information about the methods that need to be invoked on each of the components that the application is comprised of, so as to kill the complete application. Similarly the "pingApplication" method invokes relevant methods on each of the component to get information about the status of the application.

3. The Application Factory Service

The application coordinator component described above is designed to be capable of launching and coupling together the components necessary to build a distributed application and shutting them down when the user is finished. It also provides the WSDL document that defines the ensemble service. It can also provide state information about the running application to the user. It is the job of the application factory service to launch an instance of the application coordinator.

A “generic” application factory service (GAFS) is a stateless component that can be used to launch many types of applications. The GAFS accepts requests from a client that consists of the static and dynamic application information. The GAFS authenticates the request and verifies that the user is authorized to make the request. Once authentication and authorization are complete, the GAFS launches an application coordinator. The GAFS passes the static and dynamic application information to the application coordinator by invoking its createApplication method. Once the application component instances have been created, connected and initialized, the application coordinator builds a WSDL document of the ensemble application and returns that to the GAFS, which returns it to the client. When the GAFS comes up, it generates a WSDL describing itself and binds the WSDL to a registry to facilitate discovery of this service. The service maintains no state information about any of the application instances it launches.

There are two ways to make the GAFS into an application specific service. The first method is to build a web portlet that contains the necessary user interface to allow the user to supply any needed initialization parameters. The portlet then constructs the needed dynamic and static application XML documents and sends them to the GAFS to use to instantiate the application coordinator. The other method is to attach an application specific component to the GAFS which contains the application specific XML.

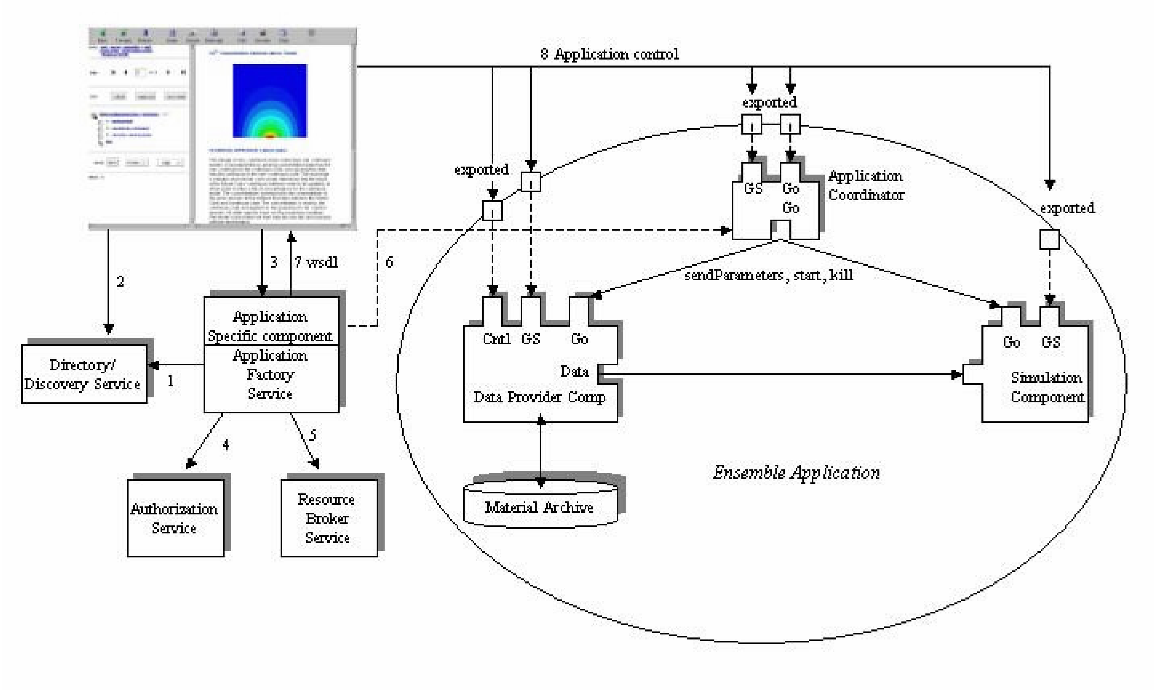


Figure 6. Complete Picture of Portal, Application Factory Service and distributed ensemble application. Each component has a GridService port (GS) and a Go control port.

The complete picture of the application factory service and application life cycle is shown in Figure 6. The basic order of events is listed below.

1. When the factory service comes up it registers itself with a directory/discovery service.
2. A web portlet can be used to discover the service.
3. The Web portlet contacts the factory service and initial configuration parameters are supplied.
4. The factory service contacts the authorization service and, if the user is authorized,
5. it contacts the resource broker which is used to compose the dynamic application information.
6. The factory service instantiates the application coordinator which then instantiates the rest of the application components and connects them. It also builds the WSDL document for the distributed application that describes the exported ports of the ensemble.
7. The WSDL is returned to the factory, which returns it to the portal.
8. Using the ensemble application WSDL, the user can contact the application and interact with it.

4. Conclusions

The prototype XCAT application factory service components described here are available for download at http://www.extreme.indiana.edu/afws. The version to be released in November, 2002 will contain the full, OGSA compliant components and contain several example service factories. The application factory service described here provides a web service model for launching distributed Grid applications. There are several topics that have not been addressed in this document. Security in XCAT is based on SSL and PKI certificates. The authorization model is based on a very simple access control list. Other, more sophisticated authorization policies are under investigation.

The other area not discussed here is the component event system. This is based on a simple XML/SOAP based messaging and event system and is available at http://www.extreme.indiana.edu/xgws. Any component may either publish or subscribe to event streams generated by other components. A persistent event channel serves as a publication/subscription target.

5. References

[1] *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration*. I. Foster, C. Kesselman, J. Nick, S. Tuecke; January, 2002.

[2] *Grid Service Specification*. S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman; February, 2002.

[3] *The Grid: Blueprint for a New Computing Infrastructure*. Ian Foster and Carl Kesselman (Eds.), Morgan-Kaufman, 1998. See also, Argonne National Lab, Math and Computer Science Division, http://www.mcs.anl.gov/globus

[4] *Legion: A Worldwide Virtual Computer*. Andrew Grimshaw. See http://www.cs.virginia.edu/~legion.

[5] *Condor – A Hunter of Idle Workstations*. M. Liszkow, M Livny and M. Mutka., Proceedings ICDCS, pages 104-111, San Jose, Ca. 1988. IEEE.

[6] *The Global Grid Forum*. See http://www.gridforum.org

[7] *Web Services Description Language (WSDL) 1.1*, W3C. See http://www.w3.org/TR/wsdl

[8] *UDDI: Universal Description, Discover and Integration of Business for the Web*. See http://www.uddi.org.

[9] *Web Services Inspection Language (WSIL)*. See http://xml.coverpages.org/IBM-WS-Inspection-Overview.pdf

[10] *Web Services Flow Language (WSFL).* See http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf.

[11] *Styles and the Design of Network-based Software Architectures.* Roy Thomas Fielding, Architectural Ph.D. Dissertation, University of California, Irvine, 2000.

[12] *Web Services for Remote Portals*, See http://www.oasis-open.org/committees/wsrp/

[13] *Design Patterns*. E. Gamma, R. Helm, R. Johnson, J. Vlissides, , Addison-Wesley, 1995

[14] *The XCAT Science Portal*. S. Krishnan, R. Bramley, M. Govindaraju, R. Indurkar, A. Slominski, D. Gannon, J. Alameda, D. Alkaire, , Proceedings SC2001.

[15] *A Component Based Services Architecture for Building Distributed Applications* . R. Bramley, K. Chiu, S. Diwan, D. Gannon, M. Govindaraju, N. Mukhi, B. Temko, M. Yechuri, *,Proceedings of HPDC*, 2000

[16] *Toward a Common Component Architecture for High-Performance Scientific Computing.*, R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, B. Smolinski, Proceedings, High Performance Distributed Computing Conference 1999.

[17] *Java Message Service Specification. Sun Microsystems Inc.* See http://java.sun.com/producst/jms/docs/html

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