Building Grid Portal Applications from a Web-Service Component Architecture

D. Gannon, J. Alameda, O. Chipara, M. Christie, V. Dukle, L. Fang, M. Farrellee, G. Fox, S. Hampton, G. Kandaswamy, D. Kodeboyina, S. Krishnan, C. Moad, M. Pierce, B. Plale, A. Rossi, Y. Simmhan, A. Sarangi, A. Slominski, S. Shirasuna, T. Thomas

***Abstract*—This paper describes an approach to building Grid applications based on the premise that users who wish to access and run these applications prefer to do so without becoming experts on Grid technology. We describe an application architecture based on wrapping user applications and application workflows as web services and web service resources. These services are visible to the users and to resource providers through a family of Grid portal components that can be used to configure, launch and monitor complex applications in the scientific language of the end user. The applications in this model are instantiated by an application factory service. The layered design of the architecture makes it possible for an expert to configure an application factory service with a custom user interface client that may be dynamical loaded into the portal.**

***Index Terms*—Grid Application, Grid Services, Portals, Web Services.**

I. Introduction

GRID technology is designed to allow users "seamless" access to applications and services running on remote resources. An example of such an application may be a tool, accessed through a web portal, which allows a user to run weather prediction simulations initialized with current conditions derived from streams of remote instrument data. In this case the data may have been initially filtered and mined by a set of service running on some remote host. The data mining services search the instrument stream for patterns that indicate bad weather. The occurrence of these patterns creates events that trigger a set of simulations running on a collection of large supercomputers which analyze various storm scenarios (This example is based upon a scenario from the LEAD project led by Kelvin Droegemeier [9]). Another example is a tool to generate computer animated movie from 3-D models of a complex molecule undergoing a folding or reaction. In this case a large scale simulation is generating periodic snapshots of the state of the molecule. These snapshots are farmed out to rendering engines to compute frames for the animation. A third example might include doing data analysis on large text or image databases that reside in remote locations. In this case the user may be looking for features cataloged in existing databases that match the features in a sample object. For example, consider the problem of understanding the spread of disease in crops by monitoring global crop production data and relating that to satellite images and climate models. The users of such a system do not want to think about how to program Grid protocols to ftp data sets and launch large supercomputer simulations. Rather, they would like to be able to pose hypothetical questions about global warming and its impact rice production in Asia.

The Grid is not yet in a state where such application scenarios are easily realized, but we are beginning to see exciting examples emerge [19,9,13,23]. In each of the scenarios described above, there are several properties that distinguish these applications as Grid applications.

1. It is very common for Grid applications to consist of a heterogeneous composition of several (and sometimes many) remote services, each of which is responsible for one part of the overall computation. Sometimes this service composition can be seen as sequences of operations that are scheduled over time as a "workflow" and other times they involve applications running at difference locations that directly interact with each other in a message-based dialog.
2. These applications often involve a complex web of collaborations (Figure 1). The end-users are scientists and engineers that only want solutions to problems and they want to do this in language of their scientific domain. They interact with the application through a web portal systems which record their choices for the application parameters. These application parameters are entered into Grid execution workflow scripts that are authored by a second group of scientists who understand how to compose Grid services into distributed applications. Domain experts are the authors and maintainers of the basic service and tool components such as the simulation codes and data filters that comprise the basic elements of the computation.

Manuscript received April 1, 2004. This work was supported in part by the U.S. Department of Energy and the NSF.
Corresponding Author: D. Gannon is with the Department of Computer Science, Indiana University, Bloomington, IN 47401 (e-mail: gannon@cs.indiana.edu).

3. The end user, who initiates or interacts with the application, sits at a remote location and will expect to authenticate his or her identity only once. From this initial authentication many resources from many different administrative domains may be accessed. The Grid security services pass the users identity to the local security mechanisms for individual services. Authorization to use the remote resources and services is provided through authorization capabilities granted by the remote resource providers and managed by an authorization service.

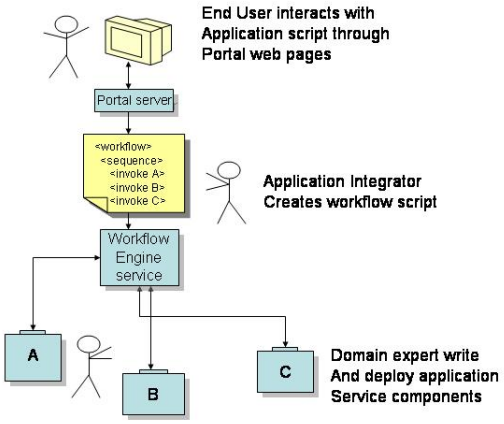

Fig. 1. Three levels of experts are involved in the design of many Grid applications. Domain experts write the individual computational components. Application integrators compose these services into distributed application. End users interact with the application through a web portal that does not expose the details of Grid management.

This paper explores the progress of an effort to construct a Grid application architecture that is based on web-services and Grid portals. This architecture is designed to build upon the still-evolving Open Grid Services Architecture (OGSA) [10], the Open Grid Computing Environment (OGCE) Portal framework [11] and service composition tools such as the Business Process Execution Language for Web Services (BPEL4WS) [4] workflow system, the Condor Dagman service [8] and distributed software component frameworks like the Common Component Architecture [2]. Very closely related ideas are found in many other projects [1,3,5,16]. This architecture does not (yet) describe a single software system that the reader can download and use (though most of the major components listed above are available). Rather it is an attempt to characterize a family of application design patterns that can be implemented with a number of existing software technologies. We will provide pointers to many of these existing technology alternatives at the appropriate point in the text.

The principle goal of this architecture is to liberate the end-user from the details of Grid middleware programming, yet still providing a means for the ambitious scientist to create new Grid applications. This is accomplished by providing a layered approach to tools that reflect the three levels of programming depicted in Figure 1. This approach also exploits the progress the commercial sector is making in transforming the success of the world-wide-web to a platform of web services designed for business-to-business transactions and intra-enterprise resource management.

The focus of this paper is on the design of a framework that allows applications to be installed and used as web services accessed through a Grid portal. We focus on two specific problems.

1. How can a Grid application be captured as a web service? In particular, how can we wrap legacy applications in simple workflow scripts that can be configured and launched by the user?
2. Once we have a way to encapsulate and execute Grid applications as Grid services, how does one provide a way for an application developer to generate an application specific interface that can be provided to the end user through the portal? In particular, how can this be accomplished without reconfiguring, or even restarting the portal server each time a new application is added?

In section II of this paper we describe the grid portal architecture with an emphasis on the way the portal server interacts with remote grid services. We address the question of generating interfaces to remote services by looking at two possible solutions. In section III of this paper we turn to the architecture of Grid Application factories and workflows. We describe in detail how an application or workflow may be encapsulated by an application factory service and combined and reused with another service.

II. Grid Portal Architecture

Figure 2 illustrated a three-layer architecture for the basic system.

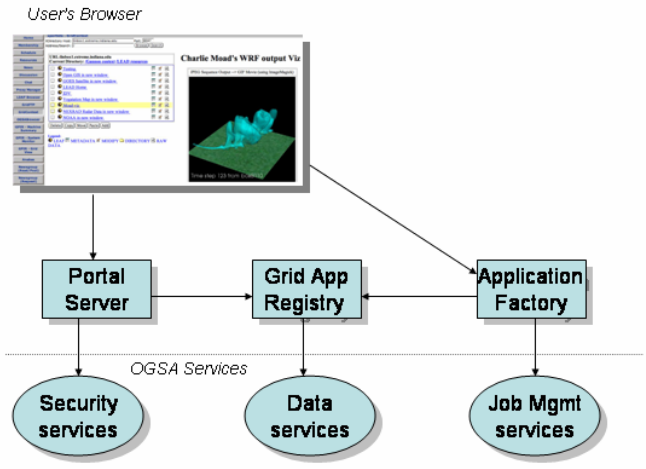

Fig. 2. The end user interacts through a web browser with a portal server based on the OGCE portal system. Specific application instances are created by an application factory which registers their existence and basic properties with a searchable Grid Application Registry. The portal server, registry and application factory interact with OGSA core services.

The top layer is the end-user who interacts with the system through a web browser. At the second layer, we have the application-level web services which include the portal server, searchable, user-level application and metadata registries, and

application factories. The third layer is the OGSA-like services that provide the means to authenticate the user, manage application event notification and mange and schedule resources. Our focus in this paper the top two layers. In the remainder of this section we will focus on the architecture of the portal server and the way in which it interacts with remote services.

The portal server used here is the Open Grid Computing Environment (OGCE) [11] portal, which is based on the Java Portlet model as implemented in Chef [6]. OGCE is being deployed in a number of Grid portal projects including LEAD [9], the NCSA alliance [18], the NSF Teragrid [25], and several other NSF, DOE and NASA projects are evaluating it. Chef is being used in the NeesGrid portal [19]. The portlet concept, which is now a Java standard (JSR-168) [14], is simple. This is the same model used by the GridSphere portal framework [12,17] and the portals available from IBM, Sun, Oracle and BEA. A portlet is a component of the web server that owns a part of the portal display window. The portlet has access to the user's session state and different portlets can communicate with each other through this mechanism. A central concept in the architecture being presented here is that the portal server provides two things:

1. A context to hold the user's session and the objects associated with that session. Some of these session objects come from the user's persistent state, which is also managed by the portal server, and some objects are created and used by portlets
2. A container for portlets which are clients to remote Grid and web services. Portlet instances within this container share the user's context. Hence a portlet which is a client to one service has access to objects created in another service.

For example, one of the OGCE standard portlets is used to fetch the user's Grid proxy certificate from the MyProxy service and store it in the user's session. Any other portlet which requires the user's proxy certificate in order to interact with a remote service on behalf of the user can fetch the proxy from the session state.

To illustrate this point in greater depth, we describe three Grid/Web service client portlets that follow this pattern.

A. *The Condor Portlet*

A good example of a portlet that interacts with a remote web service is the Condor Dagman portlet developed in collaboration with the University of Wisconsin Condor team. Condor [8] is an environment for scheduling and executing applications on distributed networks desktop computers. Dagman is a language for describing complex application workflows to be executed on Condor in terms of directed acyclic graphs. In this case, the portlet allows the user to upload descriptions of Condor jobs or workflow scripts described with the Dagman language. The user can return later and monitor the progress of the jobs. (We will return to the topic of workflow later in this paper.)

The portlet interface is shown in Figure 3 also illustrate the OGCE Chef graphical layout. The user's browsers shows a row of tabs across the top and a column of buttons on the left and a panel to the right. In this case, there is only one tab at the top and it is called "My Workspace". These tabs correspond to collaboration groups that the user has joined. Selecting a tab selects the set of portlets shared by that particular group. For example, if a user belongs to a group that is managing the data analysis tools on a Grid testbed, there may be a "data-analysis" group. If the user selects that tab for that group, he or she would see the set of portlets used by that group listed in the column on the left. These may be specialized portlets for interacting with remote data analysis grid services.

In Figure 3, the user is in his or her private "My Workspace" group and the portlets available are listed on the left. In this case the button for "Condor Portlet" has been pressed and the pane on the right displace the user interface to portlet.

Figure 3. The Condor Dagman job submission portlet. This portlet interacts with a web service that handles the actual submission into the Codor pool.

B. *The XDirectory Grid Context*

One service we use frequently is called the XDirectory Grid Context (see Figure 4). This is a secure web service that provides a simple searchable directory of a user's metadata about useful services or other data. Each entry is similar to a "resource" in the WSRF [28] sense. Security is provided by XML-signatures that are embedded in the SOAP request from the portal server to the directory service using the user's proxy credential. This is part of the WS-Security [29] model.

The client portlet is organized as two panes. In the left pane is the current "directory". If a user selects a node, the right pane displays any XHTML metadata associated with that node. Or, if the node refers to a remote Grid service of a type described

later, the user interface associated with that service is displayed. We will return to these points later.

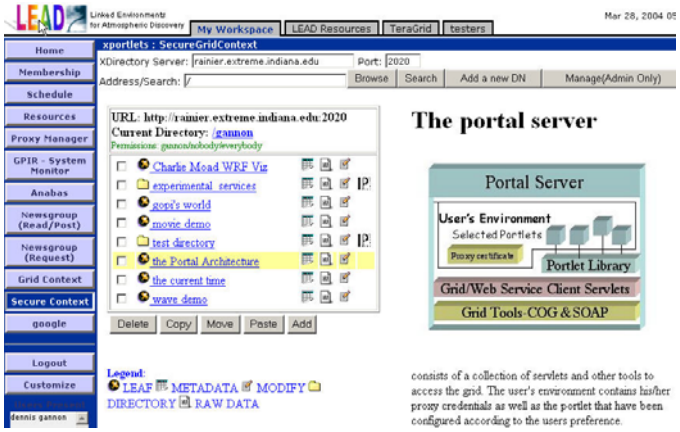

Figure 4. Secure Grid Context portlet interface to the XDirectory Web Service. The XDirectory is a secure web service that stores a tree of nodes managed by the user. Each node is either a subdirectory or a leaf. Leaf nodes contain XML metadata about that node including XHTML that is displayed for the currently selected to the right of the current directory listing. Nodes can be purely informational (such as this example which describes the portal architecture) or they can be references to other services.

C. *The OGSA-DAI Service.*

As a final example of this design paradigm, the OGSA-DAI portlet is a client to the Data Access and Integration Grid service developed in the U.K. e-Science project [20]. Portal interaction with the database through OGSA-DAI is illustrated in Figure 5. The portlet is provided with the Grid Service Handle (GSH) of the Grid Service Registry (GSR). This is done out of band. The portlet queries the registry to obtain the grid service handle to the factory, and the accompanying service document that describes the grid data service instance that has already been created. From a prior screen, not shown, the user has browsed the local files system to obtain a perform document that describes the query the user wishes to execute. That perform document is shown at the top of the portlet.

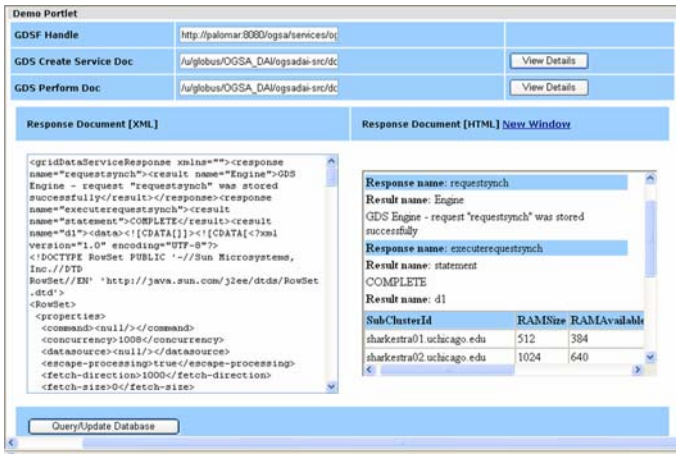

Figure 5. The OGSA Data Access and Integration Portal.

The user issues a query by selecting 'Query/Update Database' at the bottom of the page. Shown in the portlet are the results of having executed a query. Shown to the left is the response document in its XML form. On the right is the status of the execution. Partially shown to the bottom right are the results in table format. The response document has been converted to a table format using XSLT before being displayed to the user.

D. *Generic Portal Interfaces to Remote Services*

In each of the examples described above we have illustrated a portlet that provides a specialized client interface to a specific remote Grid service type. While this works, it poses a serious scalability problem. Every time a new application Grid service is created, a new portlet client must be written and configured into the portal deployment. In a more "web-like" Grid, one should be able to discover a new service and automatically load its interface into the portal server directly.

There are three approaches to solving this problem. The first, and most natural is to remember that every web service is described by a Web Service Description Language document that provides information about each port type that the service supports. This is an abstract description of the service interface. If this service is one that a human could possibly operate it should be possible to generate automatically the interface for the service. We have done this with the Xydra-OntoBrew service. As shown in Figure 6, OntoBrew is able to automatically generate a usable interface.

Figure 6. The Xydra-Ontobrew service. The directory service contains a link to the Xydra. Xydra then requests a URL for the WSDL for a service. The supplied url, http://www.xmethods.net/sd/2001/CATrafficService.wsdl, is the location of the WSDL for web service for the California Highway Traffic Condition reporter web service. Ontobrew dynamically generated the

interface shown in the top half of the figure from the WSDL. In the bottom half, we see the result of the request.

In its current form the reply from the web service is an echo of the request and the reply in raw XML form. However, it possible to do more than this. The WSDL also contains information about message replies, hence it is possible to generate a basic XHTML format for the returned values. Xydra-Ontobrew provides an automatic solution to the portlet client generation problem for simple services. However, for Grid applications that are to be presented to the scientific users, it is often desirable to build a custom interface that is richer in its interactive features. The natural solution is to build a custom client for the service that can be loaded dynamically from the service. There are a few small problems to solve. How does the portal know if a service has a special, dynamically loadable interface? And, if it has one, where is it located?

To solve this problem we can use a property of OGSI. Each Grid service has a set of attributes called Service Data Elements (SDEs) that can be interrogated by interested clients. These SDEs are similar to resources in the WSRF specification. For our Grid services that have dynamically loadable interfaces, there is an SDE for that service called, "interfaceclient". If you query for this SDE by name, you are handed the URL for the client. Theoretically, this interface client can be a Java applet, a link to an HTML document with embedded scripts that call the web service, a Java Web Start application, or any other type of interface that the Grid Service Provider wants to provide. Currently, as shown in Figure 7, we are building applications with the Applet interface in mind. As shown in Figure 7, the clients may be a Java applet.

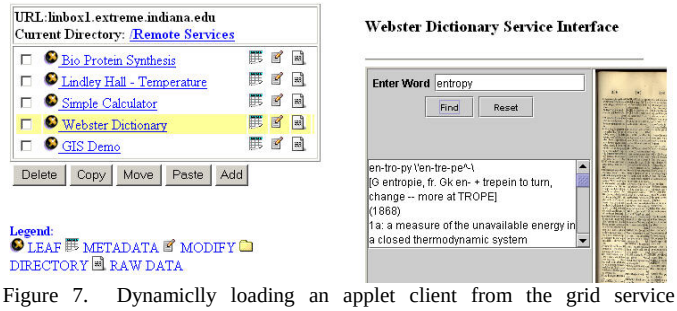

Figure 7. Dynamiclly loading an applet client from the grid service "interfaceclient" SDE. In this case the web service is an English language dictionary.

A new standard has been proposed called Web Services for Remote Portlets (WSRP) [15] which provides a very similar though, in our opinion, less flexible solution.

It should be noted that the Gridspeed project [24] provides an alternative approach to this problem. GridSpeed has an application wizard that is capable of generating interfaces to remote workflows as a single step. Though GridSpeed is not a web-service based architecture, it does show that it is possible to generate very good interfaces to complex applications with little or no programming required.

III. Workflow and The Factory Service

In a service-based model of the Grid everything is rendered as a service or a client to a service. In the general web-services world, services are stateless entities that respond to incoming messages. However, there are very few things in computing that are truly stateless. A good example of this is a workflow, which is a process that represents the automation of a sequence of interactions with a set of external services or agents. Another example is the execution of a large, long-running scientific application that consumes files and produces new ones before it terminates. While it is running, it certainly has state.

In what sense are these examples considered to be services? The answer lies in how we interact with them. For example, if we wish to be able to send messages to a running application process like "Are you almost finished?" or "please save your state and stop running" or "please start running and put you output in this file …", then it is reasonable to consider these entities to be services.

However, a more natural model is to consider *the ability* to run an application or workflow as the service and associate the actual instance of a particular execution of the application/workflow with a "resource" in a manner similar to the concept of resource as defined by the proposed WSRF specification. In this case, the resource represents the evolving state of the execution of the application as an object associated with the service that started it.

In order to turn "the ability" to run an application into a service we follow the Factory pattern and we adopt a very specific way of associating resource objects with the things created by those factories.

We begin with the assumption that each "application" is defined by a high level workflow or an execution script that invokes the underlying file management and application executables. As illustrated in figure 8, each execution of the application workflow or script is launched by a *specific application service*, which, itself, was generated by a *Primary Application Factory*. This Primary Factory is a persistent service that is used to generate the application specific services from a formal specification we will describe later. For now, we focus on the specific application instance factory.

In the model presented here each newly created service or process is associated with a new resource object that is stored in the XDirectory. These resource objects appear as subdirectories of the user context. They can be thought of as the permanent record of that service or process instance. The resource documents for the specific application instance factory contain a copy of all the application metadata used to create it as well as a reference to the service interface to access it. This is enough information to restart the service if needed.

The specific application service is specialized to launch instances of the pre-defined workflow or application script. By interacting with the supplied user interface client for the service, the user provides the missing parameter values that needed for that execution. Each executing application instance is associated with a resource document that is stored as a sub-directory of the resource object for the service that created it. This running instance resource document contains the workflow files, the parameter settings, links to outputs and a log of message events generated by the execution.

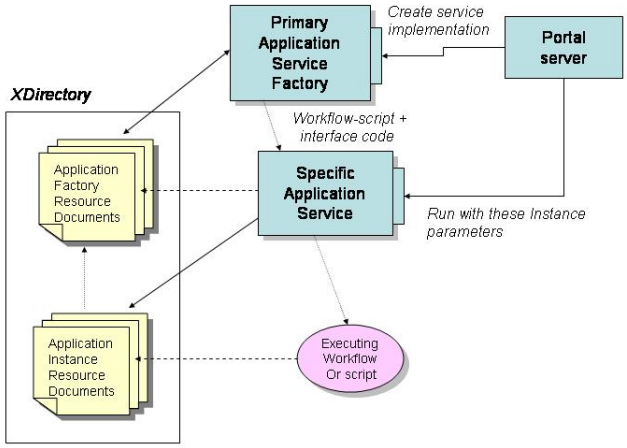

Figure 8. The application factory architecture. The generic application factory is used to generate specific factory services for specific applications. The specific application instance factories launch the specific workflow or job scripts.

To illustrate these ideas, we will consider a real workflow example. NCSA has developed a system called OGRE that extends the build tool ANT [26] to manage simple Grid workflows. An OGRE workflow is represented by an XML document that consists of a set of properties and a sequence of actions. Among the action tags are

- <rtexec> - run a program.
- <publish> - publish a message into the Grid event stream.
- <filecopy> - move a file from one location to another.

In the example illustrated in Figures 9, the workflow runs a parallel rendering program to transform output from the WRF weather simulation program into a small animation that can be viewed from a portal. The steps involved are to first boot the LAM MPI system, then execute *mpirun* with the "waterPipe" parallel renderer , then run an image converter to turn the frames into a "GIF movie", and finally copy the movie to a location visible to a web server.

```
< project name =' WRF OUTPUT Animation ' default =' run ' basedir =' . ' >
    < property environment =' env ' > </property >
    < property name =' xdpath ' value =' ${env.XDIR_PATH} ' > </property >
    < property name =' home ' value ='/u/cmoad/wrf/waterPipeOffScreen ' > </property >
    < property name =' lib ' value ='/u/cmoad/VTK/lib/vtk ' > </property >
    < property name =' images ' value ='/u/cmoad/images ' > </property >
    < property name =' dest ' value ='/u/lifang ' > </property >
    < target name =' run ' >
      < esequence >
        < rtexec args =' /usr/bin/lamboot ${home}/machinefile ' quiet =' false ' > </rtexec >
        < publish type =' MSG_EVENT ' message =' (LAM daemon booted.) ' > </publish >
        < publish type =' MSG_EVENT ' message =' (Rendering jobs begin ... ) ' > </publish >
        < rtexec args =' /usr/bin/mpirun c2,4,6,8,10,12,14,16,3,5,7,9,11,13,15,17,0 -x
              LD_LIBRARY_PATH=${lib} ${home}/waterPipe boxB110 ' quiet =' false ' >
            < outMonitor file ='OUT11 ' > </outMonitor >
            < errMonitor file ='ERR11 ' > </errMonitor >
        </rtexec >
        < publish type =' MSG_EVENT ' message = '(Rendering jobs finished.) ' > </publish >
        < rtexec args =' /usr/bin/lamhalt ' quiet =' false ' > </rtexec >
        < publish type =' MSG_EVENT ' message =' (LAM daemon halted.) ' > </publish >
        < publish type =' MSG_EVENT ' message =' (Converting images to animations) ' >
        </publish >
        < rtexec args =' /usr/local/ImageMagick-5.5.7/bin/convert -delay 10 -compress
            bzip -adjoin ${images}/boxB110/*.jpg ${dest}/wrf-boxB110.gif ' quiet =' false ' >
        </rtexec >
        < publish type =' MSG_EVENT ' message =' (Convertion completed.) ' > </publish >
        < publish type =' MSG_EVENT ' message =' (Copying the animation to the user's
              remote host through Gridftp ... ) ' > </publish >
        < filecopy dstUri =' gridftp://brick.extreme.indiana.edu/u/lifang/.hyplan ' >
            < nestedelement srcUri =' file://${dest}/wrf-boxB110.gif ' > </nestedelement >
        < /filecopy >
        < publish type =' MSG_EVENT ' message =' (All done! ) ' > </publish >
        < waitforpublisher > </waitforpublisher >
      </esequence >
    </target >
</project >
```

Figure 9. OGRE script for the parallel rendering of WRF output.

In order to monitor the progress of the rendering the script is punctuated with frequent messages published to the event stream.

The Specific Application Service is a service that contains the OGRE script in Figure 9 and is ready to create running instances when the user provides any additional needed parameters. In this case, that is the name of the WRF output file we wish to render and the storage location for the final movie. When so invoked the application services responds by doing two things. First it creates a "resource sub-directory" in the XDirectory that will contain properties of the execution. Second, it instantiates a new, transient process that is executing the OGRE script. In the XDirectory this resource node looks like a subdirectory that represents that this transient process. This is called the "Run" subdirectory for that execution. Each time we invoke this application service a new "Run" resource is added as a child to resource node for the application service (see Figure 10). This "Run" resource contains the original OGRE script and any parameters supplied to the execution, the log of all events generated by the execution and a link to the final output animation.

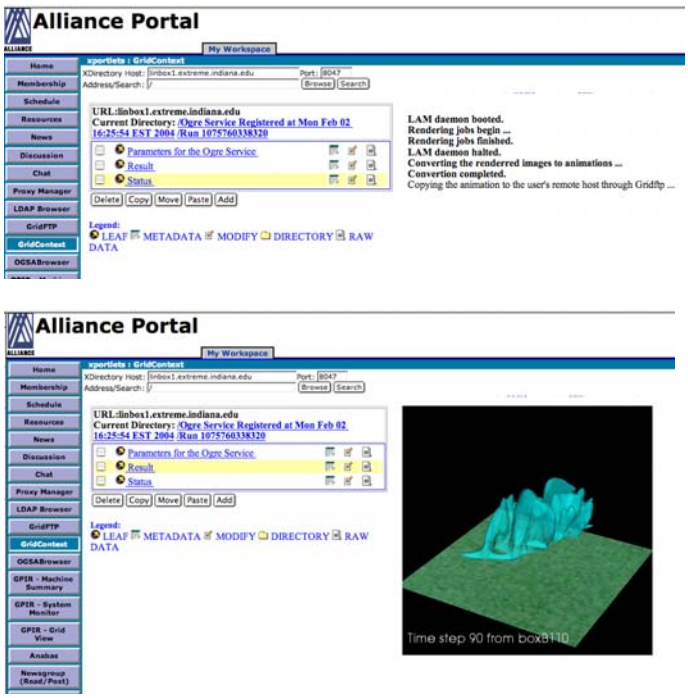

Figure 10. In the top display the user has selected the "Status" element of the Run resource for that execution. What is shown on the right is log of published messages from the execution. In the bottom display, the user has selected the "Result" element which is a link to the execution.

A. Generating a Factory Service

A Factory Service is a persistent, stateless service that allows authorized users to create instances of other services. In OGSI terms, a Factory implements the document-centric Factory PortType which is an extension of the GridService PortType. The GridService PortType provides mechanisms for discovery, life time management and notification. The Factory PortType provides the *CreateService* method for creating service instances on the Grid. It accepts as input, an XML document called the CreateServiceExtensibilityType which conforms to the OGSI specification. Clients use this document to specify the service creation parameters including the initial termination time and the set of PortTypes that are implemented by the service instance.

The CreateServiceExtensibilityType document provides all the information that the Factory needs to create a new instance of the service. But to instantiate a service, an implementation of the service must be available to the Factory. In addition, any needed implementation resource files must be made available to the factory. For example, in the case that the service being generated is the Specific Application Service which launches an OGRE execution engine, we need to supply the OGRE script, and the client used to invoke the service from the portal.

In the case where the service implementation and the client already exist and are deployed, such as with the Dictionary Service example illustrated in Figure 7, there is very little for the Factory to do. It only needs launch the service running on a specified host using a protocol like Gram or SSH.

However, in the case where the service is specific to a particular workflow or OGRE script, the Factory must dynamically generate parts of the implementation. To tell the factory what it must generate, we provide it with a ServiceMap document which is of a type that extends the OGSI's CreateServiceExtensibilityType.

The ServiceMap document has three elements; service, portType and creationParameters. The service element specifies the name of the service instance and a description of the service that is to be created. This can also be used to provide XHTML text for the portal user interface for the service. The portType specifies the WSDL portType that should be implemented by the service instance. A portType can contain several methods. Each method is mapped to an executable which is invoked when the method is invoked. The input parameters of the method are mapped to the command line arguments of the executable. The creationParameters specifies the host and port on which the service must be started and additional command line arguments.

Figure 11 illustrates the ServiceMap to create an instance of the Specific Application Service for the OGRE animation script described previously.

```
<ServiceMap>
  <service>
    <serviceName>WRF Animation Service</serviceName>
    <serviceDescription>Convert the output of a WRF job to a GifMovie
    </serviceDescription>
  </service>
  <portType>
    <portTypeName>animator</portTypeName>
    <method>
      <methodName>animate</methodName>
      <inputParams SOAP-ENC:arrayType="xsd:string[2]">
        <inputParams:string>URL of WRF output</inputParams:string>
        <inputParams:string>URL for target gif animation</inputParams:string>
      </inputParams>
      <executableName>/bin/run_ogre OGRE-SCRIPT P1 P2</executableName>
    </method>
  <creationParameters>
    <configParams name=OGRE-SCRIPT>http://xdir.iu.edu/wrf-anim.ogre
    </configParams>
    <host>rainier.extreme.indiana.edu</host>
    <port>2034</port>
  </creationParameters>
</ServiceMap>
```

Figure 11. The ServiceMap for a dynamically generated Grid service to launch the OGRE script for the animiation example.

In this case, the generated service has a single porttype with one method "animiate". The argument to this message is an array of two strings, one of which is the URL of the WRF file to animate and the other is the URL for the output animation. The executable line provides the binding from the parameters to the command line string needed to actually launch the application. The string associated with the inputParams are the names that can be used to label the parameters in the interface client However, the creation parameters, such as the host and port and OGRE script URL are the exact values that are mapped to the execution.

IV. Conclusion

This paper has presented a web-service based architecture for building specialized web services and portal clients for Grid applications. The key idea is to wrap Grid application scripts and workflows behind a Grid web service that provides the ability to run instances of the workflow or script. In many cases these specialized application service have implementations that are dynamically generated from a simple XML specification by a persistent Factory service.

Each dynamically generated object (application service or executing application instance) is associated with a resource which is stored for the user. For application execution instances the resource stores everything needed to recreate the execution including records of execution events and links to the output files.

This paper also describes a portal architecture based on the idea that communities of scientific application users would rather access the applications via web interfaces integrated into a portal. We address the problem of designing user interfaces for Grid applications that can be use in the portal and we discuss ways in which these interfaces may be included in the portal framework at runtime.

There are two topics that have not been addressed here that are very important to the architecture. One is the pub/sub notification system and the other is the Grid service authorization systems. The notification system is based on Narada [22] and WS-Notification and the authorization system is based on a Peer-to-Peer capability management framework. Both of these topics will soon be described in more detail in another paper.

Acknowledgment

The authors would like to thanks the Open Grid Computing Environment Collaboration members whose work was critical for the effort described in this paper. In particular, Gregor von Laszewski provided us with COG, and Charles Severance gave us Chef, the portal container used here. We would also like to thank Kelvin Droegemeier and Bob Wilhelmson for letting us work on the LEAD application.

References

[1] Agarwal, M., and Parashar, M. . Enabling Autonomic Compositions in Grid Environments.*Proceedings of the 4th International Workshop on Grid Computing (Grid 2003)*, Phoenix, AZ, USA, IEEE Computer Society Press, pp 34 - 41, November 2003

[2] Armstrong, R., Gannon, D., Geist, A., Keahey, K., Kohn, S. McInnes, L., Parker, S., and Smolinski, B.. Towards a common component architecture for high performance scientific computing. In *Proceedings of the The Eighth IEEE International Symposium on High Performance Distributed Computing*, 1998.

[3] Bhat, V. and Parashar, M.. Discover Middleware Substrate for Integrating Services on the Grid. *Proceedings of the 10th International Conference on High Performance Computing (HiPC 2003), Lecture Notes in Computer Science*, Editors: T.M. Pinkston, V.K. Prasanna, Springer-Verlag, Hyderabad, India, Vol. 2913, pp 373 – 382, December 2003

[4] Business Process Execution Language for Web Services Version 1.1. http://www-106.ibm.com/developerworks/library/ws-bpel/

[5] Casanova, H. and Dongarra, J, NetSolve: a network server for solving computational science problems. Proceedings SC 96.

[6] The Chef Project. http://chefproject.org/chef/portal

[7] Chipara, O., Slominski, A., Xydra – An automatic form generator for Web Services, see: http://www.extreme.indiana.edu/xgws/xydra/

[8] Condor Dagman, http://www.cs.wisc.edu/condor/dagman/

[9] Droegemeier, K.K., V. Chandrasekar, R. Clark, D. Gannon, S. Graves, E. Joseph, M. Ramamurthy, R. Wilhelmson, K. Brewster, B. Domenico, T. Leyton, V. Morris, D. Murray, P. Plale, R. Ramachandran, D. Reed, J. Rushing, D. Weber, A. Wilson, M. Xue, and S. Yalda, 2004: Linked environments for atmospheric discovery (LEAD): A cyberinfrastructure for mesoscale meteorology research and education. Preprints, 20th. Conf. on Interactive Info. Processing Systems for Meteor, Oceanography, and Hydrology, Seattle, WA, Amer. Meteor. Soc.

[10] Foster, I., Kesselman, C., Nick, J., Tuecke, S., The Physiology of the Grid An Open Grid Services Architecture for Distributed Systems Integration, www.globus.org/research/papers/**ogsa**.pdf

[11] Open Grid Computing Environment (OGCE), http://www.ogce.org.

[12] GridLab, The GridSphere Portal http://www.gridsphere.org

[13] The Grid Physics Network, http://www.griphyn.org/

[14] JSR-168 Portlet Specification. http://www.jcp.org/aboutJava/communityprocess/final/jsr168/

[15] Kropp, A., Leue, C., Thompson, R., Web Services for Remote Portlets (WSRP), OASIS http://www.oasis-open.org

[16] Matsuoka, et. al., Ninf: A Global Computing Infrastructure**,** http://ninf.apgrid.org/welcome.shtml

[17] Navotny, J. Developing grid portlets using the GridSphere portal framework, http://www-106.ibm.com/developerworks/grid/library/gr-portlets/

[18] NCSA Alliance Portal. http://www.extreme.indiana.edu/alliance/

[19] Network for Earthquake Engineering Simulation Grid (NEESgrid), http://www.neesgrid.org

[20] Open Grid Service Architecture Data Access and Integration, http://www.ogsa-dai.org.uk

[21] The Open Grid Services Infrastructure Working Group. http://www.gridforum.org/ogsi-wg, 2003.

[22] Pallickara, S. and Fox, G., NaradaBrokering: A Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids. Proceedings of ACM/IFIP/USENIX International Middleware Conference Middleware-2003. pp 41-61.

[23] The Particle Physics Data Grid, http://www.ppdg.net/

[24] Suzumura, T., Nakada, H., Matsuoka, S., Casanova, H. , "GridSpeed: A Web-based Portal Generator," to appear, Proceedings HPCAsia, 2004.

[25] Teragrid, http://www.teragrid.org/

[26] Tilly, J. and Burke, E. Ant, The Definitive Guide, O'Reilly, 2002.

[27] Tuecke, S., Czaijkowski, K., Foster, I., Frey, J. and Graham, S. Kesselman, C., Snelling, D., and Vanderbilt, P. Open Grid Services Infrastructure, Version 1.0, Global Grid Forum GWD-R, http://www.gridforum.org/ogsi-wg, March 2003.

[28] WS-Resource Framework. http://www.globus.org/wsrf.

[29] WS-Security. Web Services Security Version 1.0. http://www-106.ibm.com/developerworks/webservices/library/ws-secure/