Syracuse University
SURFACE

Electrical Engineering and Computer Science | L.C. Smith College of Engineering and Computer Science

1-1-2002

Interoperable Web Services for Computational Portals

Marlon Pierce
Indiana University

Geoffrey C. Fox
Indiana University

Choonhan Youn
Syracuse University

Steve Mock
University of California at San Diego

Kurt Mueller
University of California at San Diego

Follow this and additional works at: http://surface.syr.edu/eecs

Part of the Computer Sciences Commons

Recommended Citation
Pierce, Marlon; Fox, Geoffrey C.; Youn, Choonhan; Mock, Steve; and Mueller, Kurt, "Interoperable Web Services for Computational Portals" (2002). *Electrical Engineering and Computer Science.* Paper 81.
http://surface.syr.edu/eecs/81

This Article is brought to you for free and open access by the L.C. Smith College of Engineering and Computer Science at SURFACE. It has been accepted for inclusion in Electrical Engineering and Computer Science by an authorized administrator of SURFACE. For more information, please contact surface@syr.edu.

Interoperable Web Services for Computational Portals

Marlon Pierce and Geoffrey Fox
Community Grid Labs, Indiana University
{marpierce,cyoun,gcf}@indiana.edu
Postal Address: 501 N. Morton Street
Bloomington, IN 47404

Choonhan Youn
Department of Electrical Engineering and Computer Science, Syracuse University
and
Community Grid Labs, Indiana University
cyoun@indiana.edu

Steve Mock, Kurt Mueller
University of California at San Diego, San Diego Supercomputer Center
{mock,kurt}@sdsc.edu
Postal Address: UC San Diego, MC 0505
9500 Gilman Drive
La Jolla, CA 92093-0505

Ozgur Balsoy
School of Computational Science and Information Technology, Florida State University
and
Community Grid Labs, Indiana University
balsoy@grids.ucs.indiana.edu

Abstract

Computational web portals are designed to simplify access to diverse sets of high performance computing resources, typically through an interface to computational Grid tools. An important shortcoming of these portals is their lack of interoperable and reusable services. This paper presents an overview of research efforts undertaken by our group to build interoperating portal services around a Web Services model. We present a comprehensive view of an interoperable portal architecture, beginning with core portal services that can be used to build Application Web Services, which in turn may be aggregated and managed through portlet containers.

1. Introduction

Computing portals provide seamless access to heterogeneous computing resources through a browser-based user interface. These portals are typically built around several basic services, including job submission and monitoring, data management, and user session management. These services may be built on top of Grid technologies such as Globus [1] or SRB [2] , but this is not always the case: the Gateway portal, for example, performs job submission by direct submittal to queuing systems, while HotPage builds this service on top of Globus. The nature of web portals makes them appropriate for delivering both the aforementioned HPC-related services and more standard web-based tools (access to databases, collaboration tools, newsgroups, HTML documentation and help). In any case, the browser interface and the backend resources are separated by a middle tier that manages access to resources and communications, forming a three-tiered architecture.

0-7695-1524-X/02 $17.00 (c) 2002 IEEE

A major shortcoming of the three-tiered computing portal design is its lack of interoperability. The three-tiered architecture results in a classic stove-pipe problem: user interfaces are locked into particular middle tiers, which in turn are locked into specific back end systems. One possible solution is to define common interfaces to services and agree upon common protocols.

For portals, these common interfaces are best realized in XML, which provides a relatively simple, programming-language neutral approach. With Web Services we now have a standards-based set of tools to properly build these XML wrappings and protocols. The crucial first step is to evaluate these technologies both for standalone and interoperating portals and to document these findings.

This paper describes preliminary investigations into portal services undertaken at the Community Grids Lab at Indiana University (IU) and the San Diego Supercomputer Center (SDSC). Each group has long standing portal projects, Gateway [3, 4] and HotPage [5, 6, 7], respectively. Our projects are just two of many computing portal projects, several of which have been recently reviewed in Ref. [8]. We have undertaken the current effort as part of the Grid Computing Environment (GCE) Working Group [9] of the Global Grid Forum [10]. Services were deployed as part of the GCE testbed [11].

2. Web Services Overview

Web Services have received a great deal of attention from both the commercial and the Grid computing communities, the latter through the Globus group's proposed Open Grid Services Architecture (OGSA) [12]. Numerous overview articles describing the basic concepts of Web Services have been written (see for example [13, 14]), and we will only summarize the main concepts here.

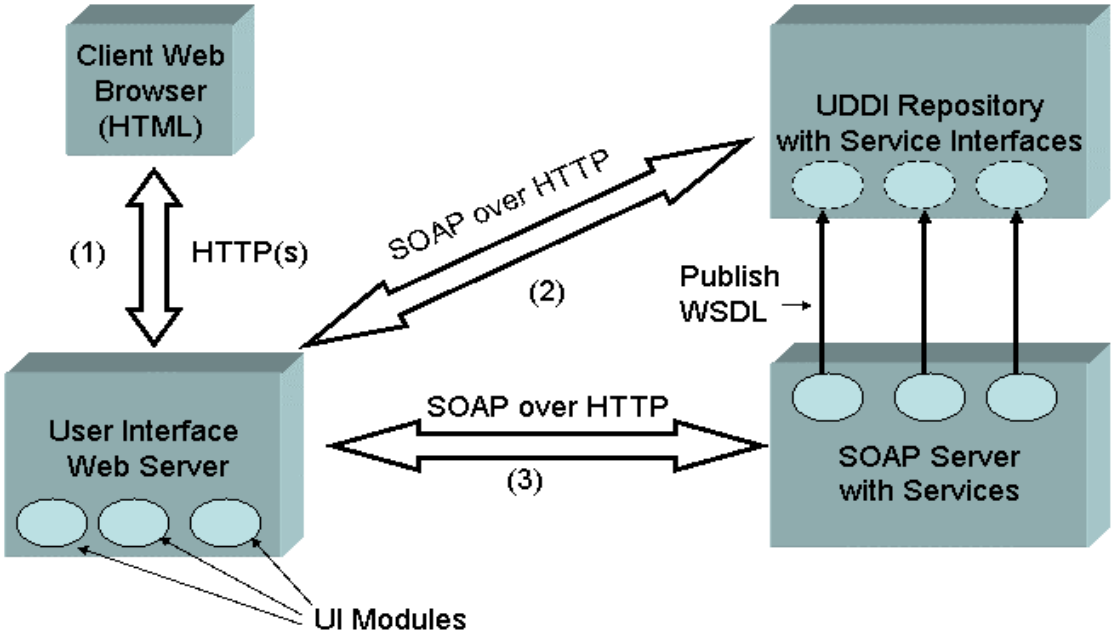

Figure 1 Basic Web Services interactions for a computing portal.

Essentially, Web Services are an XML-based distributed service system. Similar to other distributed object systems such as CORBA [15, 16], Web Services define the following concepts, with some realizations:

1. An interface definition language: Web Services Definition Language, or WSDL [17].
2. A remote method invocation protocol: Simple Object Access Protocol, or SOAP [18, 19].
3. A naming and discovery system: Universal Description, Discovery, and Integration (UDDI) [20] or the Web Services Inspection Language (WSIL) [21].

The basic interaction of the services in computing web portals is illustrated in Figure 1. A user interacts with the User Interface server, which maintains client proxies to the UDDI and SOAP Service Providers (SSP). Each of these runs on a separate web server. The UDDI maintains links to the service providers' WSDL files and server URLs. The client examines the UDDI for the desired service and then binds to the SSP. The SSP in turn acts as a proxy to some backend services, not shown, to perform a HPC task.

This approach introduces a separation between the server that manages the user interface and the server that manages a particular service. This separation is not present in the three-tiered portal model and is the key development for breaking the portal stove pipe. The User Interface server can potentially bind to any SSP. By using SOAP and WSDL universally, the portal services can be encapsulated and invoked independently of the implementation.

3. Basic Web Services for Computing Portals

The first step in our investigation is to identify a common set of services that are used by our existing portal projects. We chose to investigate the following: job submission, data management services with the Storage Resource Broker, user context management, and batch script generation. Services and clients were implemented in both Python and Java. As the first phase, we implemented job submission and file management as separate group projects and batch script generation as joint development effort, as the latter had simpler security requirements. Our next phase of development is to standardize all interfaces through the GCE and then develop secure interoperable services.

We consider the above services to be some of the basic portal Web Services. These currently define only one interface each, in WSDL, and directly implement the interface with calls to an appropriate Grid service. When building on top of the future OGSA, these portal services will have to define two interfaces: the public WSDL for using the service, and the private composition of the one or more underlying OGSA services.

As a general remark before proceeding, we note that simply using SOAP and WSDL does not automatically create interoperability. We must define properly course-grained functions that possess concise interface definitions and properly design the implementations so that they may be encapsulated in simple interfaces. The Gateway batch script generator, for example, was initially tightly integrated with the context manager and job submission services. Making this into an independent service introduced unnecessary overhead because we needed to create artificial contexts (sessions) for HotPage users. This has inspired the Gateway group to create a cleaner distinction between various internal services in anticipation of large scale Web Service adoption.

Interoperability also requires consistent error messaging. SOAP calls to services may result in both SOAP errors and implementation errors (such as, the file didn't get transferred because the disk was full). Thus the standard set of portal services that we are building must define and relay a common set of error messages for this second class of errors.

3.1. Job Submission

A job submission Web Service is a necessary component required to integrate the computational grid with Web Services. Both SDSC and IU teams built job submission Web Services but because of differing security models did not integrate their services. The SDSC team implemented this over Globus to run jobs on remote computational resources in a secure and authenticated manner, developing a Globusrun Web Service in Python. The Globusrun Web Service uses the Python implementation of GSI SOAP [21] and pyGlobus [22] to perform the submission of secure and authenticated jobs on the Grid. The Web Service exposes two different methods for job execution, one that accepts the parameters of a job as a set of plain strings and returns the results as a string, and one that accepts an XML definition of a job, and returns the results as an XML string. The DTD for the latter mechanism was designed to allow multiple jobs to be included in a single XML string and passed to the Web Service as one request. The Web Service executes the jobs sequentially, and returns the results as an XML document to the client application. The IU team implemented the SOAP job submission service as a wrapper around a client for the "legacy" CORBA [15, 16]-based WebFlow system [3]. This involved implementing a set of utility methods for initializing the client ORB, which we used to bridge between SOAP and IIOP [16]. Otherwise, the SOAP server methods wrapped the existing WebFlow methods.

To test the interoperability of different Web Services, SDSC developed a secure, authenticated Python Web Service to submit batch jobs on remote computational resources using the Grid. This simple Web Service has a method that takes string arguments that define the host and batch scheduler commands to be run, and returns a string that contains the output of the job submission. Then these string arguments are parsed, and the batch job submission Web Service uses the Globusrun job submission service previously described to submit the job. The interaction between the batch job submission Web Service and the Globusrun Web Service demonstrates a Web Service using another Web Service to perform a task.

3.2. Data Management

SDSC experimented with a SOAP interface to the Storage Resource Broker (SRB) by developing a set of Web Services in Python that expose SRB functionality. This trial was meant to explore how well Web Services could be used for data management, so a small subset of SRB's functionality was implemented as Web Services. The methods exposed in the SRB Web Services are ls, cat, get, put, and xml_call. The ls method returns an array containing the directory listing from a specified SRB collection and directory. The cat method returns a string containing the contents from a file in the SRB collection specified. The get and put methods transfer a file between an SRB collection and the client by simply streaming the file as a string. This transfer mechanism does not scale well, and was only used as a proof of concept. The xml_call method allows the client to create a single request string consisting of multiple SRB commands expressed in XML and sent to the Web Service using a single connection. The service executes the separate commands found within the requests sequentially, and returns the results as XML in string format to the client. These SRB Web Services are GSI authenticated, and use the GSI authenticated SRB command line utilities. Future work will include creating native interfaces using SOAP to the SRB server in Java.

3.3. Context Management

Gateway implements a service for capturing and organizing the user's session (or context) for archival purposes. The user can recover and edit old sessions later. We organize context in a container structure that can be mapped to a directory structure such as the Unix file system. Implementing this as a Web Service raised a couple of interesting issues. We create separate contexts for each user, and subdivide the user contexts into problem contexts, which are further divided into session contexts. Gateway modules (service implementations) also exist in contexts. This is at odds with the model, illustrated in Figure 1, in which the user interface and service implementations are logically separated into different servers with no prior relationship before the UI server sends a SOAP request. Thus we were forced to create placeholder contexts in our SOAP wrappers.

Properly implementing the stateful portal service will take further consideration. For simple services, the state simply resides on the UI server, which maintains a set of proxy clients to the SOAP services. The simple SOAP services don't need state management, but their aggregation into more complicated entities (such as a useful application) will have instances that possess state that should be preserved by a service similar to the context manager. The aggregation of distributed portlets into portals (described below) will also introduce the need for a distributed session state.

Also notable is that this service contained over 60 methods. The Gateway team may be fond of the Context Manager, but HotPage and other teams will have no use for such a complicated service. To implement this properly, the service will have to be broken up into more reasonable parts. Simply using Web Services thus does not insure interoperability. The service must be designed to have a reasonable scope and manageable interface. This requires the efforts of collaborative groups such as the GCE to determine the best practices for portals service definitions.

3.4. Batch Script Generation

The crucial test of Web Services for portals is their interoperability. SDSC and IU each converted legacy batch script generation tools into SOAP services. This effort is described in a separate publication [24]. In summary, we agreed to a common service interface, implemented it separately with support for different queuing systems, entered information into a UDDI repository and developed clients that could list services supported by each group and search for services that support particular queuing systems. Scripts could then be created through either service. Both groups implemented services in Java and tested interoperating Java and Python clients successfully. Our primary conclusions from this exercise are that SOAP and WSDL were adequate for the service's simple interface, but we need to do further tests for services using WSDL complex types, especially testing language interoperability.

We found shortcomings in UDDI for describing services that support different queuing systems. The mappings of portal groups and services into UDDI businessEntities and Services were reasonable, but UDDI lacked flexible descriptions that could be used to distinguish between something as simple as one script generator service that supports

PBS and GRD and another that supports LSF and NQS schedulers. UDDI entries are described with string comments and Identifier and Category data types based on industry standard descriptions of commercial entities. These were obviously inappropriate for our purposes. We developed workarounds with the string description, but this works only by convention. The heart of UDDI's shortcomings is that it attempts to support both human and machine clients; that is, client applications can search the UDDI but this is to collect information for human readers.

UDDI is a specialized Web Service and does not provide a general enough discovery mechanism. A more appropriate discovery system should be built around a recursive, self-describing XML container hierarchy into which metadata about services may be flexibly mapped. Possible implementations of such systems include LDAP or an XML database.

4. Secure Web Services

The core Web Services pieces described in Section 2 make no provisions for authentication, message integrity, access control, and other security concerns. As mentioned above, a GSI-SOAP implementation is available and was used by the SDSC group. However, we see the need for a general purpose way of securing SOAP that supports multiple underlying mechanisms. Access to Grid-related services is only a subset of possible portal Web Services. For these reasons, we have developed a prototype system to support Web Service single sign-on through an authentication service. We are particularly interested in supporting Kerberos [25] as a security mechanism and have used it in our prototype, but we have attempted to keep our design general and will add support for other mechanism such as PKI and Globus GSI [30].

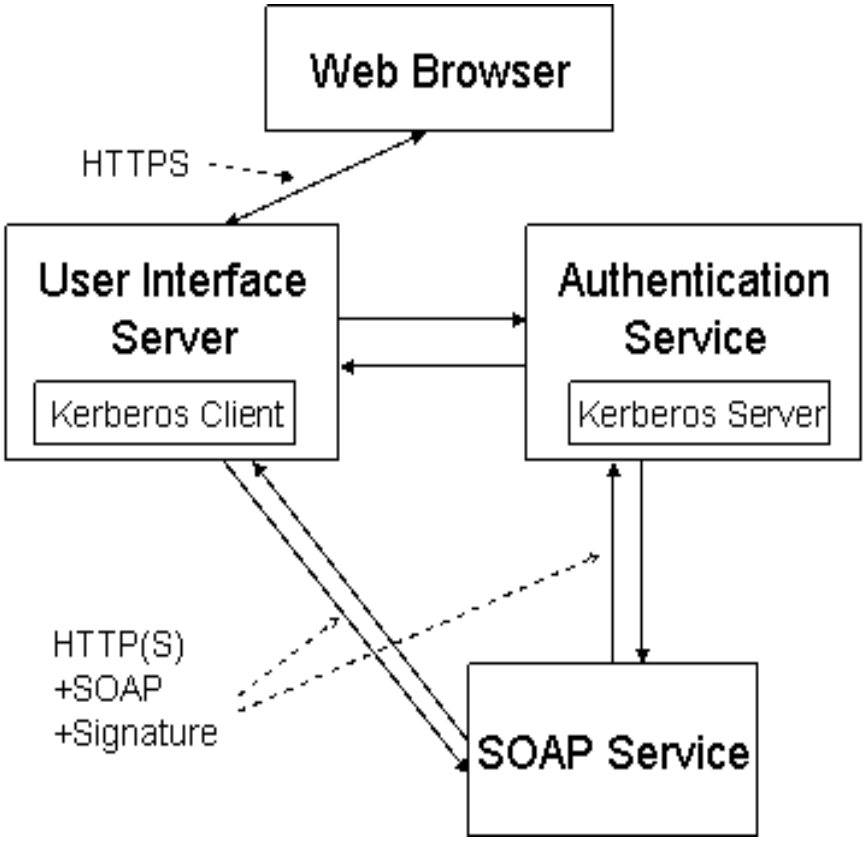

Figure 2 An assertion based authentication service using Kerberos.

Our authentication system is based on SAML, the Secure Assertion Markup Language [26]. Assertions are mechanism-independent, digitally signed claims about authentication. A SAML-consuming service can accept the signed assertion or use it to find locations of Kerberos proxy tickets or grid proxy certificates. SAML can also be used to convey access control decisions made by other mechanisms, such as Akenti [33, 34]. SAML assertions are added to SOAP messages.

We have implemented the SAML assertion specification in Java and are developing client and server applications for handling SOAP plus SAML messages. To support Kerberos, we are also developing signing methods based on the GSS API [31] *wrap* and *unwrap* methods. Such an authentication service is useful because Kerberos servers authenticate using a *keytab* file. This keytab must be kept secure and usually is readable only by privileged users. Thus we believe that limiting the use of keytabs to a single, well secured server is desirable.

Our prototype authentication system is illustrated in Figure 2 and works as follows: a user logs in through a web browser and gets a Kerberos ticket on the User Interface (UI) server. This server creates a client session object that contacts the Authentication Service, which launches a Kerberos server in a session object. The client and server then establish a GSS context, which is maintained on each server in user sessions. Each of these objects possesses one half of the symmetric key set for a particular user. Subsequent user interaction generates a SOAP request that includes a SAML assertion that is signed by the client object on the UI server. The SOAP request and assertion are sent to the desired SOAP Service Provider (SPP). The SPP does not check the signature of the request directly but instead forwards to the Authentication Service, which verifies the signature. The Authentication Service responds positively or negatively to the SPP, which may then fulfill the client's request. This procedure is the atomic step. Minimally, each server in the system would authenticate itself, and mutual authentication schemes can also be developed.

This is one specialized use for SAML. The authentication service is appropriate for authenticating to SOAP services such as a UDDI server or as a first layer for denying access to protected SOAP services. Further work needs to be done, for instance, on access control.

5. Application Web Services

The common thread among many computational web portals (including portals built with SDSC's GridPort and IU's Gateway components) is that they are science application-centric rather than service-centric. That is, the portal's primary focus is to provide support for a particular set of science or engineering codes. Basic functions such as job submission, data management, and so on are implemented in the context of the application. To cast this in terms of the current Web Services technology, the Web Services described in Section 3 are really core services that should be bound to a particular application. We thus believe the important next step is to define a general purpose set of schemas that describes how to use a particular application and bind it to the services it needs. These schemas are the foundation for what we call *Application Web Services.*

We are developing Application Web Services to provide the following features:

1. We want to be able to describe applications in a general way that is independent of the particular portal. The application description for the chemistry code Gaussian, for example, can be standard across portals.
2. In addition to defining the application interface, application descriptors also specify the core services that are required to run the application and provide context in which those services are used.
3. Having a general application description mechanism allows user interfaces to be developed independently of the service deployment. Just as clients and service implementations for the Batch Script Generation service described in Section 3.4 were developed independently by IU and SDSC after agreement to a common WSDL interface and data model, interfaces to applications can also be developed independently.
4. Client interfaces may be generated automatically from downloaded schemas using "schema wizard" techniques described below.
5. The proliferation of user interfaces to various applications (and core services) can be managed by aggregating them into component-based portlets.

In this section we describe work in progress on implementing the above features.

5.1. Application Lifecycles and Descriptors

Web Services for science applications have at least four phases of existence: (a) an abstract state, which describes options of how the application may be invoked; (b) a prepared (but not queued or submitted) instance of the application; (c) a running instance; and (d) an archived instance of a completed application run. One may propose additional states that

refine the above distinctions: the "running" state may be subdivided into queued, running, sleeping, terminating, and so on. In the context of the current discussion, however, the crucial distinction is between states (a) and (b)-(d) above. The abstract state defines the list of possible choices that can be made by the user, while states (b)-(d) and other states result from the particular choices.

Based on these classifications, we have defined the two sets of schema descriptors: the first set describes the abstract state of (a) and the second set describes the application instances in states (b)-(d). Operationally, an instance of the abstract application schema is edited by the application developer to describe how to invoke his or her application. This description is then viewed by application users, whose specific choices are instances of application instance schema.

The abstract application description is implemented as a set of three schemas: application, host, and queue. These are implemented in a container hierarchy, which applications containing one or more hosts, and hosts containing queuing system descriptions. We chose this modular approach because multiple host resource and queuing system schema descriptions exist, and we want the flexibility of adding these to our base application description.

We continue to refine our application description schema as we implement new features (such as core service bindings) and apply our descriptors to more applications. The current version of the schema is available from [35] and has the following essential elements:

1. A "basic information" element that contains information such as application name, version, and option flags.
2. An "internal communication" element that contains child elements for describing input, output, and error fields for the code. These child fields contain subfields that can be used to describe the field and bindings to particular core services needed to read or write the file.
3. An "execution environment" element that contains a list of core services needed to execute the application, along with host bindings for the particular service and host description bindings.
4. An optional, generic parameter to hold arbitrary information about the application that is not covered by the elements above.

The host binding schema contains information about the resource (such as DNS name and IP address) and all of the information needed to invoke the parent application on that resource (such as location of the executable, location of the workspace or scratch directory, and so on). In order to maximize flexibility, we also provide a general purpose "parameter" element that allows for arbitrary name-value pairs to be included. This can be used for example to specify environment variable settings needed on a particular host by a particular application. This schema also has a queue binding that contains information needed to perform queue submissions. Again, Ref. [35] gives the URL pointing to our working schemas.

The XML-based application descriptions outlined in the previous section provide a way for adding applications to a portal. Using them, the application developer can produce a portal independent view of the application. Deploying an application is done by filling out HTML forms to create an application instance. The particular application description also provides guidelines on the necessary form elements needed to create the user interface.

The particular invocations of an application by the user are stored in a separate set of schema, also viewable at [35]. These schemas are similar in form but different in intent than the application descriptions above. Instances of these schemas are used instead to contain the metadata about particular application runs: the input files used, the location of the output, the resources used for the computation, etc. Instances of these schemas form the backbone of a session archiving system, which allows users to view and edit old sessions.

5.2. Separation of Service Implementation and User Interface

The important lesson learned from the interoperable Batch Script Generation service described in Section 3.4 is that by working from a common service interface (in WSDL) and a common data model, we can independently build the client and server implementations for a core service. This idea applies equally well to user interfaces for applications. In order to do this, we need the same two pieces: the data model for the application and the WSDL interface for interacting with the data model. Here the application data model has already been defined: it is just the application descriptor detailed above. When used locally, the data model also effectively defines its own API: we convert the schema to Castor [36], which creates JavaBeans to get and set the various fields of the schema.

Converting all of the Castor methods to WSDL can be done but the resulting interface is extremely complicated and uses many complex types as arguments and in method returns, so this is not really a practical interface. Instead we are building an adapter class that encapsulates several Castor-generated get and set calls into a smaller interface definition for common tasks. These common tasks are determined from prototype user interface pages that we build that use Castor-generated classes directly.

5.3. Automatic User Interface Generation

By abstracting the application description into instances of a set of linked schema, we may automate the generation of the user interface: a web client proxy portlet can download the XML description of an application and automatically map the schema elements into visual widgets (HTML Form elements, for example). This approach can be generalized to create a general purpose schema wizard.

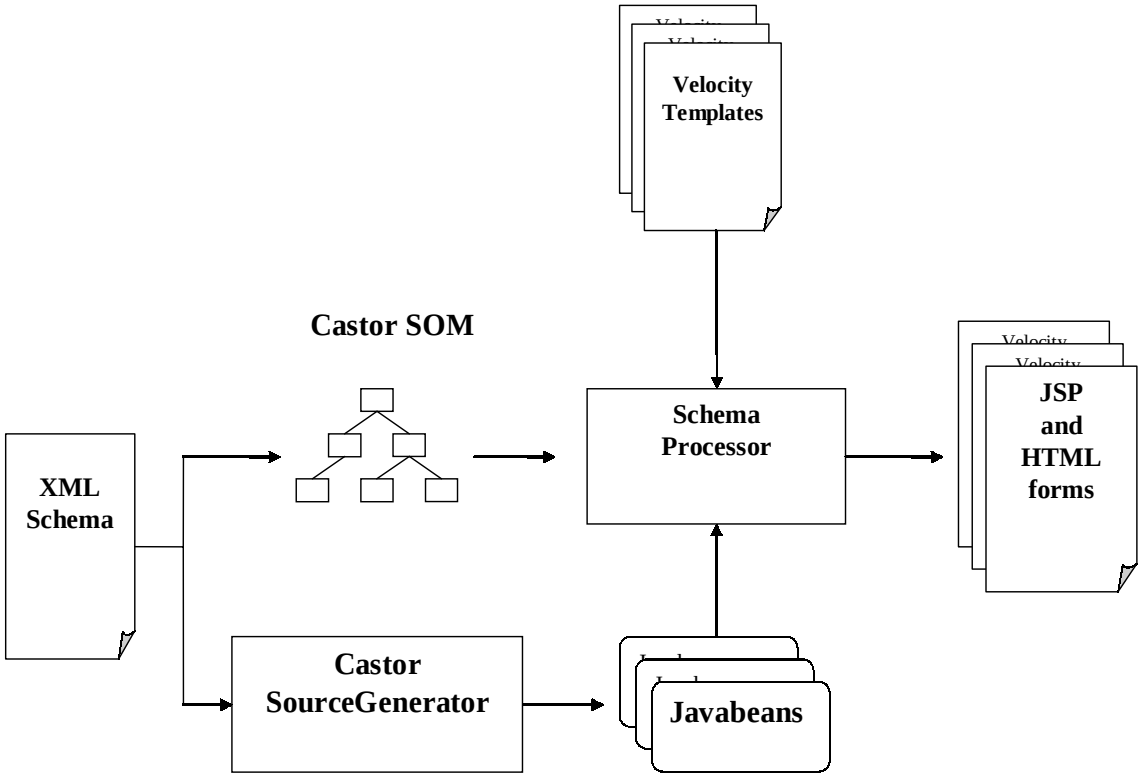

Figure 3 The schema wizard architecture maps XML schema into user interfaces and Java data objects.

The schema wizard architecture is summarized in Figure 3. A Java class (SchemaParser, here running in a JSP page on a particular server) is initialized with a URL for the desired schema and a package name for the class. SchemaParser (after validating the schema) creates an in-memory representation of the schema using Castor's Schema Object Model (SOM). The SOM provides a more convenient API for working with general schema elements than the XML DOM. SchemaParser also invokes Castor's source generator to create Java classes that are data bindings for the schema. This generates one JavaBean class per schema element. Each element comes with the associated get and set methods needed to modify element values and attributes, add or delete children, etc.

SchemaParser next compiles the generated Java data classes, deploys them as a JSP web application, and loads the new web application into the server. The necessary new directories, $TOMCAT_HOME/webapps/<project_name> for the generated JSP pages and $TOMCAT_HOME/webapps/<project_name>/WEB-INF/classes/ for the compiled classes, are created automatically.

The above steps govern the model portion of the MVC pattern, and we can also automate the view. We do this by defining JSP templates (in Velocity) for several different schema constituent types: single simple types, enumerated simple types, unbounded simple types, and complex types. The SOM is used to transverse the schema to detect if the element corresponds to one of the templated types above. As types are detected the Velocity engine is started and used to create a JSP page with the appropriate property values obtained from the SOM. The template also sets the necessary Java package imports needed to use the Castor classes. The template also provides the action logic that connects the Castor classes' get and set methods to the HTML form elements. Each template generates a JSP nugget that is used to build up the final page using the <%@ include … > directives. The resulting JSP page has form elements that can be filled out to create an instance of the schema. The resulting Java object can be marshaled back to a XML instance of the given schema and saved on the file system with some name. Old instances can be read in and unmarshaled to fill out the form elements.

5.4. Portlets

The previous sections dealt with separation of user interface from service implementation for applications. This user interface may even be automatically generated, as in the schema wizard scenario. It is possible now to provide one or more interfaces for each application. Thus a computational web portal may be built up out of user interfaces to the application interfaces as well as interfaces to core services such as file transfer or job monitoring that may interest a user.

Technology already exists for aggregating these user interfaces. One popular, freely available, open source application is Jetspeed [32], and commercial products implementing similar ideas are also available. Generally, portlet systems possess the following features:

1. Portlet types exist to retrieve both local and remote web content. Each component web page is contained in a table and the final composite web page is a collection of nested HTML tables, each containing material loaded from the specified content server.
2. In the case of remote web content, the portlet is a proxy that loads the remote URL's contents and converts it into an in-memory Java objects.
3. Portal administrators decide which content sources to provide. In Jetspeed, this is done by editing an XML configuration file (local-portlets.xreg) to extend the appropriate portlet.
4. Users can customize their portal displays by decorating them with only those portlets that interest them.

Thus for example a particular portlet could contain application interfaces for structural mechanics, chemistry, physics, and fluid dynamics applications, but each individual user's interface consists only of the interfaces that interest him.

It is possible to build specific portlets that are local to the portal server that can load schema and generate interfaces in a manner described in the previous section on the schema wizard. However, we do not favor this approach. We instead want to keep user interface development distinct from the portlet container, using the latter only for control and display. This approach allows us to build a few general purpose portlets that can load remote content and preserve legacy user interfaces.

To this end, we have written a general purpose portlet that extends Jetspeed's simple WebPagePortlet. This portlet (called WebFormPortlet) is basically a proxy to a remote web site. Like WebPagePortlet, our portlet loads a remote web site and keeps an in-memory copy for reformatting. We have also implemented some additional features:

1. The portlet can post HTML Form parameters.
2. The portlet maintains session state with remote Tomcat servers.
3. The portlet remaps URLs in the remote page, so that the content of pages loaded from followed links and clicked buttons is loaded inside the portlet window.

These features are enough to support remote application interfaces. For example, the legacy Gateway user interface implementation consists of several linked web form pages that maintain session state between the pages. With the above WebFormPortlet features, these pages can be loaded and navigating inside a Jetspeed container running on a separate server.

6. Summary and Future Directions

We have presented a comprehensive view of computing portal architectures based around Web Services, with a description of the initial investigations into the core services, interoperability issues, and security that will form the basis for an interoperable and interchangeable Web Service architecture for computational portals. Going beyond these services,

we discussed the need for application-specific services and data models that can be used to encapsulate entire applications independently of the portal implementation.

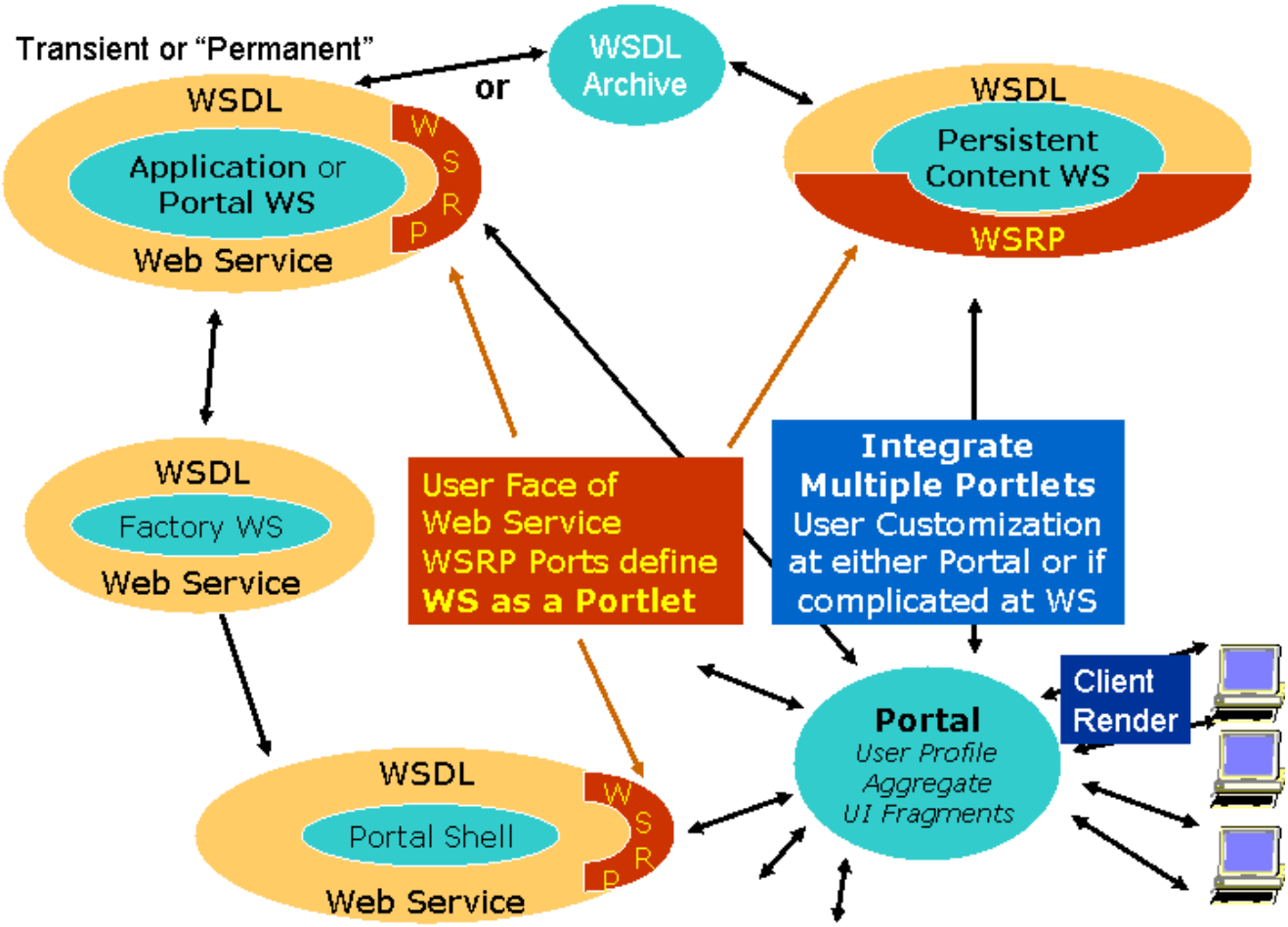

Figure 4 Schematic representation of a comprehensive service-based portal architecture.

We believe that the technologies discussed in this paper, while independently interesting, need to be integrated into a common architecture. Figure 4 schematically illustrates a possible architecture that illustrates these ideas. We believe that the integrated architecture begins to resemble a distributed operating system: user interactions are through a finite list of basic commands that operate in a "shell" or execution environment. These commands encapsulate "system" level calls to actually interact with computing resources. The point is that there are two levels of interfaces: one to the system level Grid infrastructure and one to the core portal service. The user does not interact with the system level services but instead through a tool chest of core services, which in turn interact with the system level interfaces.

Three types of portal Web Services are depicted in the figure: applications, portal shell commands, and content (information) services that interact with XML data objects. Each has a WSDL (or perhaps OGSA) definition and is implemented out of system level Grid services. The portal shell services are basic services such as those we have described in Section 3. These services may be bound to specific resources through a factory creation process, such as discussed in Ref. [37]. These services may be useful to the user by themselves, or they may be composed into specialized applications. One may envision a scripting environment for example that provides the syntax for linking the various core services (redirecting output through pipes, for example) and the logic for executing services. The application descriptors discussed in this paper represent one way of aggregating core services for the specific purpose of describing scientific applications.

Finally, all of the portal services (core, application, information, and others) define interfaces that separate the service implementation from the client implementation. The client implementation is created from the published interface and may either be static or dynamically generated (as described in Section 5.3). These client interfaces themselves can be aggregated into a portal interface. The discovery, binding, and communication between such portlet components may be handled through standards such as the WSRP [29].

7. References

[1] Foster, I., and Kesselman, C. Globus: A Toolkit-Based Grid Architecture. In *The Grid: Blueprint for a New Computing Infrastructure*, Foster I., and Kesselman, C.,eds. Morgan Kauffmann, 1999.

[2] Buru, C., Rajasekar, A., Wan, M. The SDSC Storage Resource Broker. Proc. CASCON '98 Conference, Toronto, Canada, 1998.

[3] Akarsu, E., Fox, G., Haupt, T., Kalinichenko, K., Kim, K., Sheethalnath, P., and Youn, C. Using Gateway System to provide a desktop access to high performance computational resources. Proceedings of the Eight IEEE International Symposium on High Performance Distributed Computing, August, 1999.

[4] Pierce, M., Youn, C., and Fox, G. The Gateway Computational Web Portal. To be published in Concurrency and Computation: Practice and Experience.

[5] Mock, S., Thomas, M., and von Lazewski, G. The Perl Commodity Grid Toolkit. To be published in Concurrency and Computation: Practice and Experience.

[6] Dahan, M., Mueller, K., Mock, S., Mills, C., Regno, R., Thomas, M. Application Portals: Practice and Experience. To be published in Concurrency and Computation: Practice and Experience.

[7] Thomas, M. P., Mock, S., Dahan, M., Mueller, K., Sutton, D., The GridPort Toolkit: a System for Building Grid Portals. Proceedings of the Tenth IEEE International Symposium on High Performance Distributed Computing, August, 2001.

[8] Concurrency and Computation: Practice and Experience. Forthcoming special issue on Grid Computing Environments.

[9] Grid Computing Environments. Accessed from http://www.computingportals.org.

[10] Global Grid Forum Website. Accessed from http://www.gridforum.org.

[11] Thomas, M. P., et. al., The GCE Interoperable Web Services Testbed. Submitted to Special Issue of the Journal of Parallel Distributed Computing: Computational Grids, R. Wolski and Jon Weissman, co-editors (2002). Available at: http://www.computingportals.org/testbed. http://www.webservicesarchitect.com/content/articles/modi01.asp.

[12] Foster, I., et al. The Physiology of the Grid: an Open Grid Services Architecture for Distributed Systems Integration. Accessed from http://www.globus.org/rearch/papers/ogsa.pdf.

[13] Vaughan-Nichols, Stephen J.. Web Services: Beyond the Hype. Computer. Vol 35, No. 2, p 19.

[14] Wolter, Roger. XML Web Services Basics. Accessed from http://msdn.microsoft.com/library.

[15] Common Object Request Broker Architecture Web Site. Accessed from http://www.corba.org.

[16] Siegel, J. ed. CORBA 3: Fundamentals and Programming. John Wiley and Sons, 2000.

[17] Web Services Description Language (WSDL) 1.1. Accessed from http://www.w3c.org/TR/wsdl

[18] Simple Object Access Protocol (SOAP) 1.1. Accessed from http://www.w3c.org/TR/SOAP

[19] Seely, S. SOAP: Cross Platform Web Service Development Using XML; Prentice Hall: Upper Saddle River, 2002.

[20] Universal Description, Discovery, and Integration (UDDI). Accessed from http://www.uddi.org.

[21] Web Services Inspection Language (WS-Inspection) 1.0. Accessed from http://www-106.ibm.com/developerworks/webservices/library/ws-wsilspec.html. See also http://msdn.microsoft.com/library/default.asp?url=library/en-us/dnglobspec/html/ws-inspection.asp

[22] GSI-SOAP was developed by the Distributed Systems Department at Lawrence Berkeley National Laboratory.

[23] Jackson, K. pyGlobus: a Python Interface to the Globus Toolkit. To be published in Concurrency and Computation: Practice and Experience. Accessed from http://www.cogkits.org/papers/c545python-cog-cpe.pdf.

[24] Mock, S. et al. A Batch Script Generator Web Service for Computational Portals. Proceedings of Communications in Computation, International Multiconference on Computer Science, 2002.

[25] Neuman, B. C. and Ts'o, T. Kerberos: An Authentication Service for Computer Networks, IEEE Communications, 1994.

[26] SAML Documents and specifications are available from http://www.oasis-open.org/committees/security/.

[27] Application Metadata Working Group web site. http://ecs.erc.msstate.edu/AMD-WG/index.jsp.

[28] Fox, G., et al. Collaborative Portals for Earthquake Science. To be published in Concurrency and Computation: Practice and Experience.
[29] Web Services for Remote Portlets. Accessed from http://www.oasis-open.org/committees/wsrp/.
[30] Foster, I., Kesselman, C., Tsudik, G., and Tuecke, S. A Security Architecture for Computational Grids. ACM Conference on Computers and Security, 1998.
[31] Generic Security Services Application Program Interface, Version 2, accessed from http://www.rfc-editor.org/rfc/rfc2078.txt.
[32] Jetspeed Overview. Accessed from http://jakarta.apache.org/jetspeed/site/index.html.
[33] Thompson, M., Johnston, W., Mudumbai, S., Hoo, G., Jackson, K., and Essiari, A. Certificate-based Access Control for Widely Distributed Resources. Proceedings of the Eight Usenix Security Symposium. 1999.
[34] Akenti: Distributed Access Control. Accessed from http://www-itg.lbl.gov/Akenti/.
[35] Pierce, M., Youn, C., and Fox, G. Application Web Services. Available from http://www.servogrid.org/slide/GEM/AWS.doc. Schemas are also available from http://www.servogrid.org/GCWS/Schema/index.html.
[36] The Castor Project Home Site. Accessed from http://castor.exolab.org/.
[37] Gannon, D., et al. "Grid Web Services and Application Factories." Draft paper available from http://www.extreme.indiana.edu/xgws/afw/appFactory.pdf.