Building Problem Solving Environments with Application Web Service Toolkits

Choonhan Youn[a,b], Marlon E. Pierce[b], and Geoffrey C. Fox[b]

[a]Department of Electrical Engineering and Computer Science, Syracuse University
cyoun@ecs.syr.edu

[b]Community Grid Labs, Indiana University
501 N. Morton Street, Suite 224
Bloomington, IN 47404-3730
{marpierc,gcf}@indiana.edu

Abstract

Application portals, or Problem Solving Environments (PSEs), provide user environments that simplify access and integrate various distributed computational services for scientists working on particular classes of problems. Specific application portals are typically built on common sets of core services, so reusability of these services is a key problem in PSE development. In this paper we address the reusability problem by presenting a set of core services built using the Web services model and application metadata services that can be used to build science application front ends out of these core services, and the management of multiple versions of services.

Keywords: Web services, Application toolkit, computing portal, portal interoperability, negotiation

1. Introduction

Web browser-based scientific portals provide the user-centric view of computational grids [1]. Building on a foundation of core services such as job submission, file management, and session management, we can build sophisticated, domain-specific Problem Solving Environments (PSEs). Numerous such portals/PSEs have been developed, with varying degrees of specialization to applications. Some examples include NASA's Information Power Grid, San Diego Supercomputing Center's Hotpage and its application-specific spin-offs, Pacific Northwest National Laboratory's Ecce system, UNICORE, and our own Gateway project. References for these and other projects may be found in [2] and [3].

An important problem that must be addressed by PSE developers is the reusability and interoperability of their constituent core service components. Obviously, PSE developers want to reuse their own core service implementations to build new portals. We may go a step further and recognize the need for sharing reusable services between PSE development groups. Our experience has also shown that many of the basic services (such as batch script generation for queuing systems) are reinvented by many different groups [4]. A much improved process would be to build all portal services with well-defined interfaces and remote method invocation through a commonly accepted messaging system. Although consensus about common accepted definitions for interfaces, not to mention runtime interoperability, is hard to achieve between multiple groups, identification and reuse of another group's particular service tool from a common repository is a realistic goal, provided that agreement may be reached on how to plug these services into one's PSE. One powerful approach is to use XML for interface definitions and messaging to facilitate implementation independence. Web services [5] and the Open Grid Service Architecture [6] provide the specific XML languages for these mechanisms.

In this paper we describe some specific core applications built in the Web service framework that can be used to build Problem Solving Environments out of reusable parts. We then address two important information requirements of these services: application metadata and version negotiation. Application metadata forms the basis for Application Web Service toolkits, which allow new scientific applications to be built out of core services. Version negotiation is a prototypical approach for solving the problems of service compatibility.

2. Web Service-Based Computing Portal Architecture

Most computing portals are based on a three-tired architecture and thus have a classic stove-pipe problem in aspects of services. In order to integrate distributed services, the computing services should be designed for the interoperability and reusability. For addressing these challenges, we present a Web service based computing portal architecture around Web

services model which have emerged as a popular standards-based, and service-oriented framework for accessing network-enabled applications. Web services have an XML-based, distributed computing paradigm to address the heterogeneous distributed computing services. It defines a technique for describing a service component, accessing a service, and discovering a service.

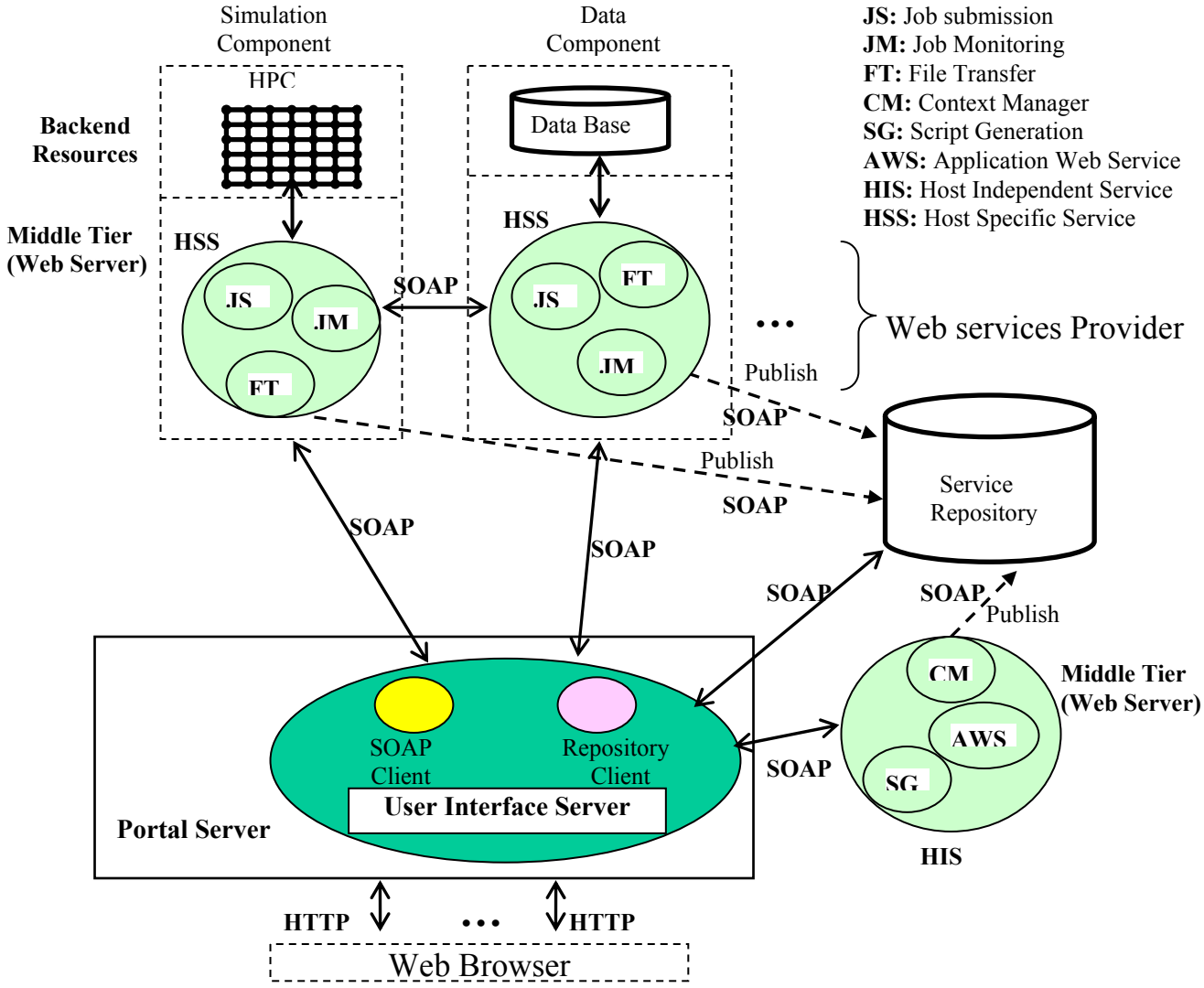

Fig. 1. Architecture of Web service-based computing portals.

For reference we provide the following brief summary of the major constituents of Web service systems. More extensive descriptions can be found in Ref. [5]. Service description is provided by the Web Services Description Language (WSDL) [7]. Message invocation may use (at least in part) the Simple Object Access Protocol (SOAP) [8] for remote procedure calls. Information services and registries may use the WS-Inspection Language (WSIL) [9] and/or the Universal Description, Discovery and Integration (UDDI) [10] service. Web services are loosely organized, and we do not imply that the exclusive use of all these languages or services defines Web services.

The Fig. 1 illustrates the architecture for this kind of Web services system from the point of view of a portal. The basic point is that common interfaces can be used to link different multi-tiered components together. In this figure we have two distinct backend hosts, a high performance computing host and a database host, perhaps implemented by different groups. These hosts run various services that interact with the host to execute operating system command, etc. Information about these services is maintained by one or more information services. The user interacts with the service hosts and information

servers indirectly, through client proxies maintained by the User Interface Server (UIS). The UIS is responsible for aggregating the various core services into application-specific PSEs. The various component interfaces may be collected as *portlets,* as described in [4], which define how user interface components can be plugged in and managed by portal administrators and users. Jetspeed [11], for example, is an open source portlet container system. Client stubs can bind and access these services with the protocol and mechanism prescribed in the service description by first contacting the service repository, UDDI that maintains links to the Web service Providers' WSDL files and server URLs and finding a service to use. In this architecture, the control layer between the server that manages the user interface and the server that manages a particular service becomes decoupled. This separation makes it possible to provide the interoperable (or at least pluggable) services.

3. Core Web services for Computing Portals

From Fig. 1, we may classify services as either host-specific services (HSS) or host-independent services (HIS). HSS includes job submission, job monitoring, and file management service. Instances of these services are bound to particular hosts. HIS include context management, script generation, and application services. HIS services are usually information/metadata services that are not tied to specific service points; i.e. the service provided does not depend on the location. We consider the above to form a basic set of portal Web services. In general, these services must be chosen so that they define properly course-grained functions with concise interface definitions. We now present several examples of such services.

3.1. Job submission

Computational portals must obviously allow users to execute scientific applications. We have defined a WSDL interface for executing commands on specific hosts systems (see Appendix, item 1). This service is remotely accessed through SOAP messages over HTTP. The service may execute operating system calls directly or may interact with Grid services through client APIs. We implement this service in Java and typically but typically use it to run external (non-Java) commands. We usually couple this service with the batch script service described below.

3.2. File Manipulation

Portal users must be able to move files between their desktops and various backend destinations, as illustrated in Fig. 1. They must also be able to manipulate remote files transparently. We have defined Web services for such file management, allowing users to transparently move, rename, and copy files on remote back-ends. Files may also be transferred between desktop and backend, and cross-loaded between different backend sites. The full service interface may be obtained from the URL in the Appendix (item 2).

File uploading and downloading services illustrate the use of SOAP messages with attachments [12] in the RPC messaging style. SOAP attachments are non-XML files that are appended to the SOAP message and are useful for sending binary data and files with known MIME formats. For example, the file uploading service sends SOAP messages with attachments encapsulated in a MIME multipart format from the desktop to the SOAP server. We implemented file uploading and downloading service using the DataHandler class from the JavaBeans Activation Framework [13]. This class is used to represent arbitrary binary data (which could include text files, binary documents, and program data files). This provides a consistent interface to data available in many different sources and formats. Apache Axis [14] provides the serializer and deserializer for the DataHandler class so that a SOAP client (in Fig. 1) may send SOAP messages with attachments to a remote SOAP server that is running the file management Web service. This service must still implement the file management details described in the WSDL interface for the received file.

We further allow the SOAP client to instruct one service to move a file to another service directly. We refer to this as cross-loading. For example, in Fig. 1, a user may request that a file be transferred from the data storage component to the simulation component. The file management service implementation has the capability of also acting as a SOAP client to another service. Here the data component service receives a cross-load message from the SOAP client and uses this to construct a new SOAP upload request to the simulation component.

3.3. Context Management

The Context Management service archives interactions with the computational portal. Each user is assigned a unique set of context data, which is used to store all information from the user session. The default context data contains "Date", "LastTime", "Descriptor", "Directory", and "CurrentChild". In general, context data can be used to store any useful metadata. For example, we define user session data using application instance metadata (described below), but context data can be used to store the location of this XML file. Context data can later be recovered and edited by users to, for example, modify old sessions in order to resubmit jobs.

In our terminology, a context is a container that can hold an arbitrary number of string name-value pairs, as well as other contexts. These contexts are defined as a recursive XML schematic diagram shown in Fig. 2 (see example at URL given in item 3 of Appendix), so the schema supports an arbitrarily deep and complex tree-shaped data structure. In practice, a user's context data consists of a root context, with child contexts for particular problem groups for that user and grandchild contexts for particular problem sessions.

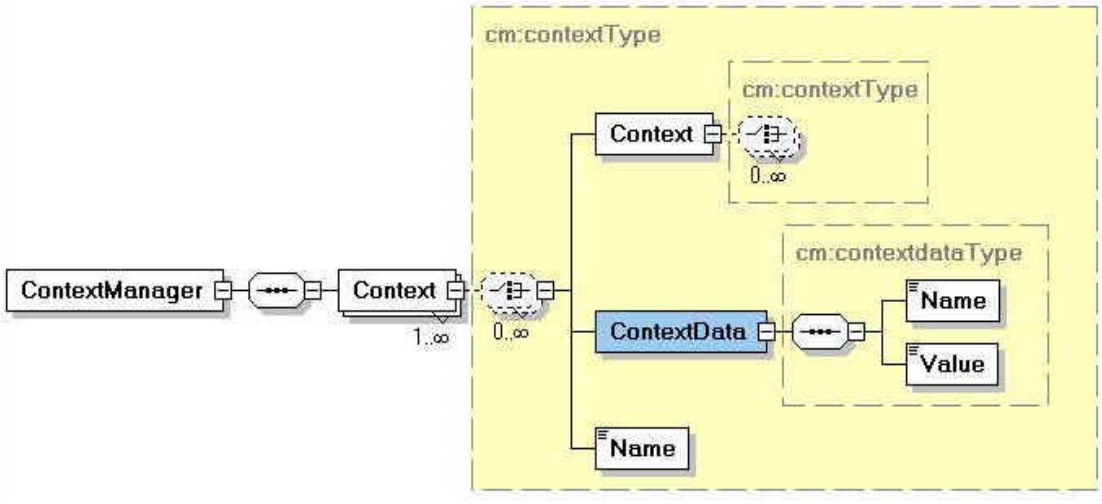

Fig. 2. Schematic diagram for the Context Manager XML descriptor.

The context data service is built over the data model described above. We refer to the actual service interface for manipulating the context data as the Context Manager. This is defined in WSDL and exposes the following methods for manipulating context data:

1. A user can add one or more contexts according to the problem domain and sessions and so on. The context path should follow UNIX path. For example, the current path for a user context is empty character (""). So, if a user has "test" context for the problem domain and want to add new context, "session" under that context, the context path should be used "test/session". Using the JXPath [15] which implements XPath, the context path is checked whether or not the context is available.
2. A user can search the context data which is stored in a user context during the portal interaction using XPath queries and store the context data which is the input from the portal interaction for editing and managing in a specified context using the context path.
3. A user can remove the specified context including all of contexts and the context data in a user context using the context path.
4. A user can get the list of "children" list of that context for reviewing the context data, using the context path.

The WSDL for the entire interface may be obtained from the URL given in the Appendix (item 4). The above method calls are used for internally manipulating the context data. Internally, the context data instance is represented as a set of data classes created with Castor [16]. We store these schema instances persistently either on the file system (mapping context data nodes to directories) or in an XML-native database such as Xindice [17].

We described above how to implement a Context Manager Web service but not how to use it. Application Web Services (described below) give the user various choices and are used to generate forms needed to collect the information needed to run the code. The user's particular choices constitute a separate XML document, the Application Instance Descriptor. This contains all the metadata about a particular invocation of the application. Using the Context Manager Web service, a user

can add the session context in a user container structure that can be mapped to a directory structure for storing this user application instance XML file. So, a user can review and edit these data from the session context which contains all of information. From the user's Application Instance XML document, the user's job script which contains the queuing script such as PBS and the user script which the user run on the application code is created by the Script Generator Web service. This job script for a user is stored by Context Manager Web service which maintains the user's information. When a user submits the job, a user gets the location of the user's job script by the Context Manager Web service running on some particular host.

3.4. Script Generation

We have developed a Batch Script Generation (BSG) Web services for users who are unfamiliar with high performance computing systems and queue schedulers. The WSDL interface may be obtained from the URL given in the Appendix (item 5). This service assists users in creating job scripts to work with a particular queuing system such as the Portable Batch System (PBS). From our experience, most queuing systems were quite similar and could be broken down into two parts, a queuing system-specific set of header lines, followed by a block of script instructions that were queue independent. Queue scripts are actually generated on the server, based on user's choice of machine, application code, memory requirement, input/output file name and parameter, etc. This information is stored as an XML document, the Application Instance Descriptor. The BSG service accepts this XML document as an argument and generates the queue script of the requested type. The structure of the BSG service implementation allows it to extensibly support different queuing systems.

A previous version of this service has been described in Ref [18]. We have extended this service to integrate it with the Application Instance schema described below. The latest version returns both the generated batch queuing script and the shell script needed to submit the queue script to the queue of a specified host. All information needed to generate these scripts is obtained from the Application Instance data. Actual submission uses the Job Submission service described above. Scripts may be moved to the appropriate host using the File Management service.

3.5. Job Monitoring

Job monitoring services may be built in one of two ways: periodic client polling and server event notifications. We currently use the polling method, which we prefer for reliability, but have built event-style prototypes based on email notifications.

The polling job monitoring Web service makes one method available for use by clients through a SOAP RPC for monitoring the execution of a job running in a queuing system. The WSDL interface may be obtained from the URL given in the Appendix (item 6). Basically, a batch job submission returns a unique job identifier that can be used for enquiry about the job status. If the job is submitted to a batch scheduler it is in the pending state while sitting in the queue waiting to be executed. The job is active when actually executing and may become suspended due to pre-emption mechanisms. In case of normal completion the job status is done, otherwise the job is failed.

The input to this method is the user account name and the scheduler type, such as "PBS". The service implementation is designed as a factory so that support for particular schedulers can be added in a well defined way. If the scheduler type is not supported by the Web service, then a message to the effect is returned to the client. If the scheduler is supported, then the user name is passed to a handler created for that specific scheduler. The scheduler handlers are custom-written methods that generate a WSDL complex type, effectively an XML data object given the user name and return the array of the generated a WSDL complex type that contains the job status of the scheduler.

4. Application Web Services and Toolkits

The core services described above are intended to serve general purposes. These must be organized in a more meaningful fashion for use in PSEs. In particular, we have developed a set of XML schemas for describing scientific application metadata. Here "application" means specifically some science code on the computational grid. Given that the application has been installed on some hosts, we want to describe how this application may be added to a PSE.

We want to specifically enable two different classes of PSE users: application managers and application users. Application managers are responsible for adding and managing the user interfaces for applications, while application users are scientists or engineers who wish to use the application. Given these two classes of users, application metadata descriptor data models come in two sets: abstract application descriptors for managers and application instance descriptors for end users. The descriptor components and rationale behind their structure is described in [4]. The URLs for these schemas are given in the Appendix (item 7).

For internal reference we review the schema structure briefly. Abstract application descriptors consist of one or more applications (schema complex types), which are described by various fields such as application name and version, application flags and flag formats. We also include elements that describe the Web service bindings for the applications input, output, and error. A particular application is also described by the Web services needed to run the application on a particular host. The service bindings are in turn bound to service hosts. Supplemental host information (such as host name and IP address, queuing system, standard working directories, and so on) are described is complementary schemas. We have attempted to design these schemas to be modular, so that third party host and queue/scheduler schemas may be plugged into the system.

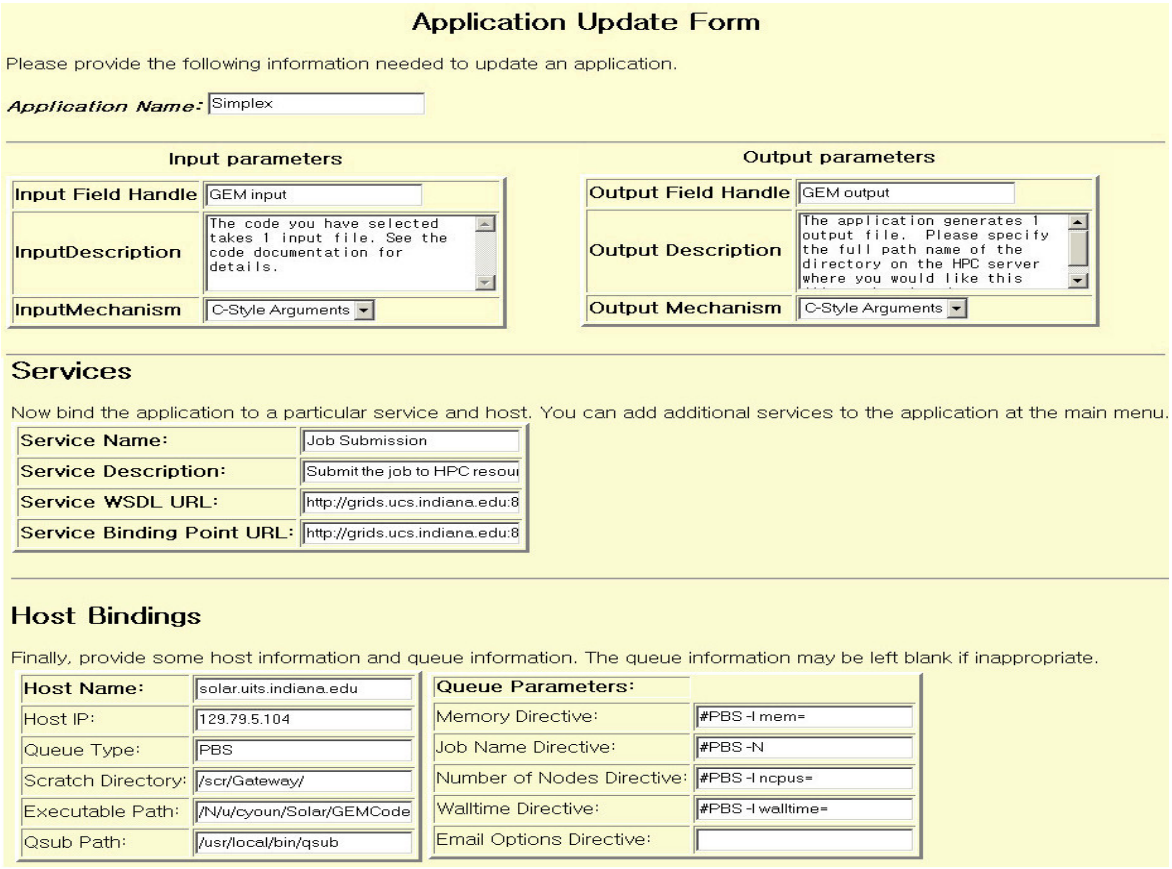

Fig. 3. Sample application administration view for the application, Simplex.

The application metadata must now be turned into a useful service. The schemas themselves can be mapped to Java language bindings automatically using tools such as Castor [16]. The schemas are too complicated, however, to be used to define a useful WSDL interface, so we instead implement "Façade" wrapper classes that simplify access to the schemas, at the loss of some functionality. The wrapper interfaces are still quite large, and the URL for the wrapper interface definition is in given in the Appendix (item 8).

Clients and user interfaces may be built out of these interfaces. Fig. 3 illustrates part of an interface for the application manager. Application managers may provide various pieces of information needed to create instances of Abstract Application Descriptors. The illustrated form indicates the simple case of adding an application with one input and one

output field, no command line flags, etc. The application can then be bound to a set of services on various hosts for submission and file transfer. The form elements are mapped to client stubs, which in turn (via SOAP) can be used to view and modify remote AWS schema instances.

Information provided by the application manager (which services may be used for file input and output, which may be used for application execution and so on) is used to generate user interfaces for application users. One example of this is shown in Fig. 4, which is a screen shot of a PSE for earthquake modeling built out of portlets. The Fig. 4 illustrates a form generated for the user for submitting a particular application (a finite element code, in this case) to the indicated queuing system, based on an Application Descriptor instance.

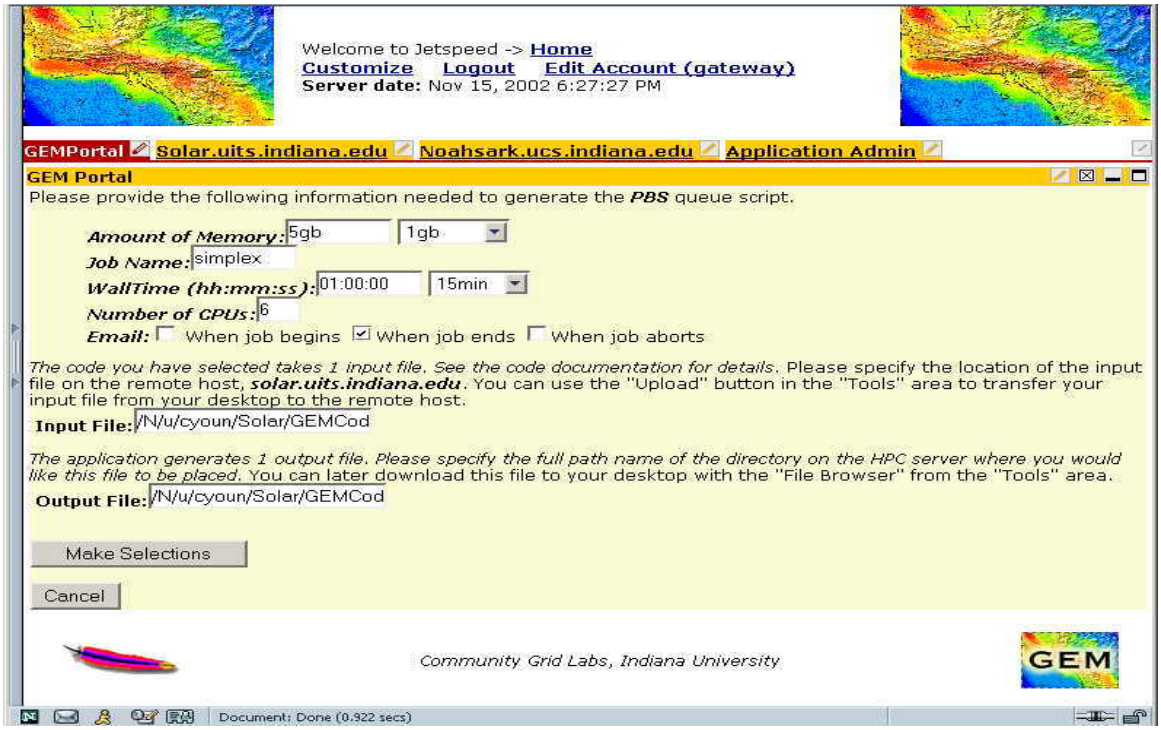

Fig. 4. Sample generated user view of application (Simplex).

Application metadata is intended to serve as a supplemental, specialized data model that is more appropriate for Application Portals than WSIL or UDDI. We believe this has an important role in computational services. The Open Grid Service Infrastructure (OGSI) specification [28] provides an important mechanism for describing particular service metadata. On top of this we need service classification systems, which organize WSDL definitions in meaningful ways. Such classification schemes are subjective and not likely to be fully standardized, so we anticipate a future interesting problem of managing many different application metadata ontologies.

In addition to the application schema, we define the host and queue binding description schema which is modular container as our design mechanism. The host binding schema contains three elements about resources: First, the host information element such as DNS name and IP address; Second, the application information element need to invoke the actual application code running on that host such as location of the executable code, location of the workspace or scratch directory; Third, queue information element such as the location of the executable queue command running on that host and the queue type which the host supports. Like the application schema, for maximizing the 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. The queue binding schema contains the queue script information (the job name, user account name, memory size, the number of CPUs, the wall time, the email options and the general purpose parameter for the queue) based on the queuing system, such as PBS. These schemas are designed to be plugged into the application descriptor schemas, and may be replaced by other schema definitions.

5. Web Service Negotiation

Distributed service systems such as we are building will need to address many issues as they move from demonstrable prototypes to production systems. One crucial problem is negotiation of quality of service. A classic example here is a file transfer service that has different bindings for default usage, high performance, and reliability, respectively. Service compatibility is a related problem: assuming the model depicted in Fig. 1, the client must first determine if it has the appropriate stubs to invoke the discovered service. Alternatively, the client and server may support various versions of a particular service and will want to negotiate the optimal version between them.

We briefly review here some motivating examples that we considered when designing a prototype system for Web services. The *Secure Socket Layer (SSL) handshake protocol* [19], [20] is used to negotiate the cipher suite first and then begins with an exchange of information between the client and server. That is, the goal of the SSL Handshake between the client and the server is to negotiate on an acceptable protocol version such as v2 or v3, and to select the appropriate set of cryptographic algorithms, i.e., cipher and hash methods, and to authenticate uni- or bi-directionally using PKI certificate, and to securely distribute shared secrets for exchanging the data.

The *Session Initiation Protocol (SIP)* [21], [22] is a standard of Internet Engineering Task Force, especially for Voice over IP and an application-layer control protocol that can establish, modify and terminate multimedia sessions or calls. There is a need to negotiate a multitude of parameters, settings, and algorithms for example, compression algorithms, encryption algorithms, code book size, and message integrity mechanisms when setting up sessions using SIP. This negotiation would take place prior to session establishment, between any two SIP entities. The result of the negotiation is a key that will be used in subsequent transactions to maintain the negotiation state. This key is carried as a SIP header ("Key" field) in further SIP messages.

These two protocols are dissimilar in application but alike in their implementation of negotiation: both use an offer/answer approach [23]. This model may be readily extended to Web services. We suggest that SOAP over HTTP is the appropriate protocol for the negotiation phase, regardless of the actual protocol used for actual service invocation. In this model, one participant in the session generates a SOAP message that constitutes the offer. For example, an offer might include a set of desired protocols and interface versions that the offerer wishes to use. The offer is conveyed to the other participant, called the answerer. The answerer responds with a SOAP message to the offerer. The answer indicates whether the service request is accepted or not, and if accepted, the selected parameters from the offer. This offer/answer model is most useful in Web service negotiation where information from both participants is needed for negotiating some parameters.

We may implement Web service negotiation through extensions to WSDL. In particular, a given WSDL interface can contain multiple portTypes. For example, multiple versions of the same service may be specified in separate portTypes in the same WSDL document. URLs for examples are given in the Appendix (item 9). In our prototype system, WSDL descriptions that support negotiation between various portTypes must define one additional, specialized portType for negotiation. Before using a particular Web service such as job submission, the WSDL PortTypes that describes the service should be taken with the WSDL negotiation PortType. Web service negotiation is an additional "PortType" and Operation that provide interfaces for creating negotiation service.

We implement Web service negotiation as a supplemental portType (in a separate name space) that is included in the WSDL service description as an extension. Given the wide range of possible uses, we have attempted to make this modular, which would allow "standard" negotiation descriptions to be plugged into the portType frame work. The schema for this portType can be obtained from the URL given in the Appendix (item 10). The negotiation portType contains two elements: "operation" and "parameters". The "operation" element is intended to be extended by another URI that defines a standard negotiation message format. The "parameters" element is extended to contain the actual data used in the negotiation.

We suggest the approach is for all Negotiation operation schemas to have a hierarchical URI beginning with the name space similar to http://.../Negotiate. For example, the schema defining the standard format for file transfer protocol negotiation may be named http://.../Negoatiate/.../FTProtocol. A similar naming system can be proposed for other uses, such as version negotiation for a particular service. The schema here defines the format for describing protocols. The actual protocols available for negotiation from a particular service are contained in the extension to the "parameters" element.

As a testing example for negotiation Web service, we describe family of parameter schemas for "standard" types of negotiation which is applicable to the negotiation descriptor: Version picking (namespace: http://.../Negotiate/../Version) and Protocol picking (namespace: http://.../Negotiate/../Protocol). Each of these has a schema that extends basic parameter

schema provided by the negotiation descriptor. Version picking schema contains a Version service name and a set of Version values.

The basic interaction of a Web service client with services for version control is as follows. For managing the operation messages between participants, we should pick a parameter family defined by a URI such as http://.../Negotiate/Version. A client sends its parameters configuration and the URI of that configuration to the service provider. The service provider makes a decision when receiving the client's parameter list and URI for the negotiation information. The service provider selects the particular version based on any desired choosing algorithm. The service provider then sends the chosen version back to the client.

We are currently implementing a prototype negotiation system using SOAP for handling negotiation messages between client and the desired service. This requires modifications to the "call" and "service" methods of the Apache Axis [14], which is SOAP engine for combing negotiation method with a computing Web service, for example, the job submission Web service.

6. Conclusion and Future Directions

In this paper, we have presented the design and implementation of several core portal services and Application Web Services. These form the basis of PSEs, and our emphasis has been on the development of reusable services that can form the basis for multiple PSEs. We have identified and classified the kinds of services depending on the service deployment. Based on this classification, the portal developer can construct specific implementations and composites of primitive service components and can also provide services that may be shared among different portals. We have demonstrated application-specific services and data models that can be used to encapsulate entire applications independently of the portal implementation. As we discuss elsewhere [4], the current infrastructure provides the service interoperability and reusability.

Next, we will consider some specific extensions (security, negotiation, job composition) to the architecture of a Web service based computing portal. First, Secure Web services will be considered that we need secure SOAP messages between user interface server and the repository and the service provider for the user authentication, based on the message-level security architecture. SOAP security should be provided through standard interfaces to independently specific mechanisms such as Kerberos [24], PKI [25]. The general approach is to use the assertion based security such as SAML [26], WS-Security [27] into SOAP messages. An assertion, for example, SAML, WS-Security, is an XML document describing the information about authentication acts performed by subjects, attributes of subjects and authorization decisions, created with a specific mechanism.

References

[1] I. Foster, C. Kesselman, The Grid: Blueprint for a New Computing Infrastructure, Morgan Kauffmann, 1999.

[2] Concurrency and Computation: Practice and Experience, forthcoming special issue on Grid Computing Environments. Abstracts and links to accepted papers available from http://aspen.ucs.indiana.edu/gce/gce2001index.html.

[3] F. Berman, G. Fox, T. Hey, Grid Computing: Making the Global Infrastructure a Reality, Wiley, London 2003.

[4] M.E. Pierce, et al, Interoperable Web Services for Computational Portals, Proceedings of Supercomputing 2002. Available from http://sc-2002.org/paperpdfs/pap.pap284.pdf.

[5] Graham, S. et al, Building Web Services with Java, SAMS, Indianapolis, 2002.

[6] I. Foster, 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.

[7] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, Web Service Description Language (WSDL) version 1.1. W3C Note 15 March 2001. Available from http://www.w3c.org/TR/wsdl.

[8] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H.F. Nielsen, S. Thatte, D. Winer, Simple Object Access Protocol (SOAP) 1.1. W3C Note 08 May 2000. Available from http://www.w3.org/TR/SOAP/.

[9] K. Ballinger, P. Brittenham, A. Malhotra, W. A. Nagy, S. Pharies, Web Service Inspection Language (WS-Inspection) 1.0. IBM and Microsoft November 2001. Available from http://www-106.ibm.com/developerworks/webservices/library/ws-wsilspec.html.

[10] T. Bellwood, L. Clemont, D. Ehnebuske, A. Hately, M. Hondo, Y.L. Husband, K. Januszewski, S. Lee, B. McKee, J. Munter, C. von Riegen, UDDI Version 3.0 Published Specification. 19 July 2002. Available from http://uddi.org/pubs/uddi-v3.00-published-20020719.pdf.

[11] Apache Jetspeed Project. Available from http://jakarta.apache.org/jetspeed/site/index.html.
[12] J.J. Barton, S. Thatte, H.F. Nielsen, SOAP Messages with Attachments. W3C Note 11 December 2000. Available from http://www.w3.org/TR/SOAP-attachments.
[13] JavaBeans Activation Framework. Available from http://java.sun.com/products/javabeans/glasgow/jaf.html.
[14] Apache Axis. Available from http://xml.apache.org/axis/.
[15] JXPath. Available from http://jakarta.apache.org/commons/jxpath/.
[16] The Castor Project. Available from http://castor.exolab.org/.
[17] XML database, Xindice. Available from http://xml.apache.org/xindice/.
[18] S. Mock, et al, A Batch Script Generator Web Service for Computational Portals. Proceedings of Communications in Computation. International Multiconference on Computer Science, 2002.
[19] Java Secure Socket Extension (JSSE) Reference Guide. Available form http://www.ssw.uni-linz.ac.at/Services/Docs/JDK/guide/security/jsse/JSSERefGuide.html.
[20] A. Freier, P. Karlton, P.C. Kocher, The SSL Protocol Version 3.0. Internet Draft (1996). Available from http://wp.netscape.com/eng/ssl3/3-SPEC.HTM.
[21] M. Handley, H. Schulzrinne, E. Schooler, J. Rosenberg, SIP: Session Initiation Protocol, IETF Request for Comments 2543 (1999). Available from http://www.ietf.org/rfc/rfc2543.txt.
[22] S. Parameswar, B. Stucker, The SIP Negotiate Method, Internet Draft, Internet Engineering Task Force, June, 2002 Work in progress. Available from http://www.softarmor.com/sipwg/drafts/draft-spbs-sip-negotiate-01.txt.
[23] J. Rosenberg, H. Schulzrinne, An Offer/Answer Model with SDP, Internet Draft, Internet Engineering Task Force, February 21, 2002 Work in progress. Available from http://rfc3264.x42.com/.
[24] Kerberos: The Network Authentication Protocol. Available from http://web.mit.edu/kerberos/www/.
[25] Understanding PKI. Available from http://verisign.netscape.com/security/pki/understanding.html.
[26] B. Blakley, S. Cantor, M. Erdos, et al, Bindings and Profiles for the OASIS Security Assertion Markup Language (SAML), OASIS 10 January 2002. Available from http://www.oasis-open.org/committees/security/docs/draft-sstc-bindings-model-09.pdf.
[27] B. Atkinson, G. Della-Libera, S. Hada, M. Hondo, P. Hallam-Baker, et al, Web Services Security (WS-Security) Version 1.0. IBM 05 April 2002. Available from http://www-106.ibm.com/developerworks/webservices/library/ws-secure/.
[28] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maquire, T. Sandholm, D. Snelling, P. Vanderbilt. Open Grid Services Intrastructure (OGSI) Version 1.0. Draft paper, April 5, 2003. See http://www.gridforum.org/ogsi-wg/drafts/draft-ggf-ogsi-gridservice-29_2003-04-05.pdf.

Appendix: URLs for Schema Definitions and WSDL Interfaces

The schema definitions and interface definitions for the services described in this paper are available from the following URLs:

[1] Submitjob WSDL interface. Available from http://www.servogrid.org/GCWS/services/Submitjob?wsdl.
[2] File Transfer WSDL interface. Available from http://www.servogrid.org/GCWS/services/FileService?wsdl.
[3] Context Manager Schema. Available from http://www.servogrid.org/GCWS/Schema/Cmhtml/cm.html.
[4] Context Manager WSDL interface. Available from http://www.servogrid.org/GCWS/services/ContextManager?wsdl.
[5] Script Generation WSDL interface. Available from http://www.servogrid.org/GCWS/services/ScriptGenerator?wsdl.
[6] Job Monitor WSDL interface. Available from http://www.servogrid.org/GCWS/services/Jobmonitor?wsdl.
[7] Application Web Service Schemas are available from http://www.servogrid.org/GCWS/Schema/index.html. These are also described in the following internal report: Pierce, M., Youn, C., and Fox, G.: Application Web Services. Available from http://www.servogrid.org/slide/GEM/Interop/AWS.doc and http://www.servogrid.org/slide/GEM/Interop/AWS2.doc.
[8] Application Descriptor WSDL Interfaces. Available from http://www.servogrid.org/GCWS/services/ApplicationDescriptor2?WSDL and http://www.servogrid.org/GCWS/services/ApplicationDescriptor3?WSDL.
[9] Negotiation Web service example. Available from http://grids.ucs.indiana.edu:8001/GCWS/negwsdl/sjneg.wsdl.
[10] Negotiation Web service Schema. Available from http://www.servogrid.org//GCWS/Schema/Negohtml/negotiation.html.