Interoperable Web Services for Computational Portals

Marlon Pierce, Choonhan Youn, and Geoffrey Fox
Community Grid Labs, Indiana University
{marpierce,cyoun,gcf}@indiana.edu
Postal Address: 501 N. Morton Street
Bloomington, IN 47401
Phone for Corresponding Author: (812) 856-1212

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
Phone for Corresponding Author : (858) 534-8398
FAX: (858) 822-5407

Abstract

Computational web portals are designed to simply access to diverse sets of high performance computing resources, typically through a computing Grid interface. An important shortcoming of these portals is their lack of interoperability 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 basic portal services that can be used to build application web services, which in turn may be aggregated and managed through distributed portlet containers.

1. Introduction

Computing portals provide seamless access to heterogeneous computing resources through a browser-based user interface. These resources are typically high performance computing (HPC) or Grid computing resources, but the nature of web portals makes them appropriate for delivering both HPC-related features (file transfer, job submission and monitoring, HPC resource usage monitoring) and more standard web tools (access to databases, shared white boards for collaboration, HTML documentation and help). 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.

Computing portals have 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.

A major shortcoming of 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 are best realized in XML, which provides a relatively simple, programming-language neutral approach. With Web services (WS) 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. The efforts of several other portal research groups are described 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]. Web services have been extensively reviewed (see for example [13, 14]), and we will only summarize the main concepts here. Essentially, Web services are an XML-based distributed object 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 for distributed objects: Universal Description, Discovery, and Integration (UDDI) [20] or the Web Services Inspection Language (WSIL) [21].

The basic interaction of the services 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 is natural for Web services, and 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 potentially be encapsulated and invoked independently of the implementation.

3. Basic Web Services for Computing Portals

We studied the following set of services: 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 separately and batch script generation collectively 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, 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 of 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 for HotPage users.

Interoperability also requires consistent error messaging. SOAP calls to services may result in both SOAP errors and implementation errors (the file didn't get transferred because the disk was full). Thus the standard set of portal services that we are building toward 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 have not yet integrated 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 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. 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 introduce the need for a distributed session state.

Also notable is that this server 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 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 [23]. 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 between portal groups and services and 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 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. At the heart of the UDDI shortcoming 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 [24] as a security mechanism and have used it in our prototype, but we will add support for other mechanism such as PKI and Globus GSI [30].

Our authentication system is based on SAML, the Secure Assertion Markup Language [25]. 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 have also developing signing methods based on the GSS API [31] *wrap* and *unwrap* methods. We will find authentication services useful because Kerberos servers authenticate using a *keytab* file. This keytab must be kept secure and usually is readable only by root. 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 is 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.

The above 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. Future Work

5.1. Application Web Services

Computing portals often support specific science applications running on various machines. An important issue that we must address is the following: a science application exists at some place. How do we add this code to the Grid?

We may think of the basic web services in analogy with Unix shell commands, and in fact the Unix shell provides a reasonable starting list of services: directory listings of files on machines with various options, commands to view file contents, get system information, and so on. Essentially, though, the core services are not suitable for public consumption. Instead, they must be composed into application web services.

We must also account for the application developer who simply wants to have a well defined way of adding a particular science application to a grid. To do this we need two interfaces: one that composes the application out of basic web services and one that defines how the application is to be accessed and used. The former is related to workflow and has been conceptually described in Ref [23]. The latter defines what we have previously called an application descriptor, and Ref [26] contains a rudimentary version of this.

One may ask how to distinguish between core services and application services. As a rule of thumb, the application service has a transient state and an archival afterlife. The core services on the other hand do not need state. Their instances exist only for the length of the request/response from the User Interface server.

5.2. Application Web Service Life Cycles

Science application web services have three phases of existence: abstraction, running instance, and completed instance. Abstract services are described through an application descriptor, mentioned above. Once the service is requested, it spawns a specific Web service instance, created through an appropriate factory for the requested resources. This concept has been described in terms of OGSA in Ref. [12]. After the service terminates, it exists in a final, archival state. This is illustrated in Fig. 3.

The archived state contains all the metadata describing that particular invocation: where the process ran, what input files where used, where the output files where stored, and so on. This application metadata is not itself a web service but will be accessed through application metadata Web services that must be defined [27].

5.3. Portlets and Web Service Shells

As mentioned above, each application web service defines a public interface (in XML) that describes how it is to be accessed by clients. Thus the application web service is separated from the user interface and the two parts can be developed independently, by different groups. We may now think of the user interfaces to applications as being components themselves, or "portlets" that can be aggregated into a collective portal. One may think of the portal as supplying the equivalent of the Unix shell. The user invokes a particular service through a portlet shell, which spawns an instance to a service that uses specific resources.

These portlets can thus be described with XML interfaces, communicate with other portlets via SOAP, and be published and discovered in a distributed portlet registry. Java-based portlet systems such as Jetspeed [32] are currently available and we are modifying them to support more sophisticated content. The WSRP specification [29] represents the next step in portlet development and we will be pursuing this.

6. Summary

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 and security that will form the basis for the full system. We have identified several areas for further research.

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. Submitted to the 2002 International Multiconference on Computer Science.
[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/.

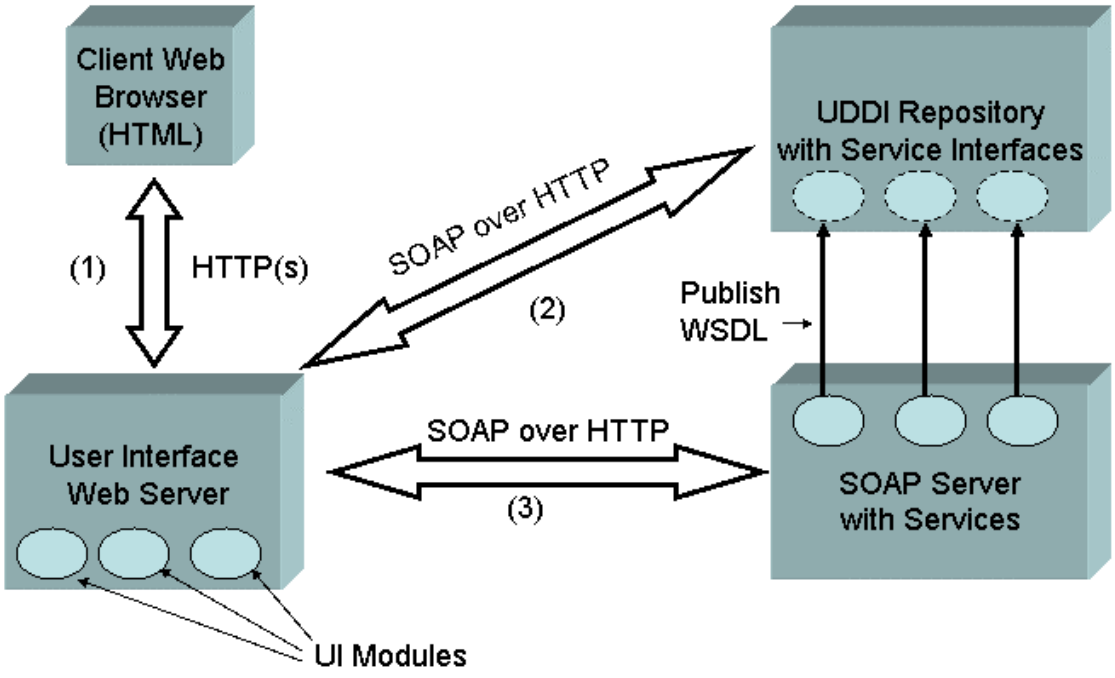

Figure 1 Basic web services interactions for a computing portal.

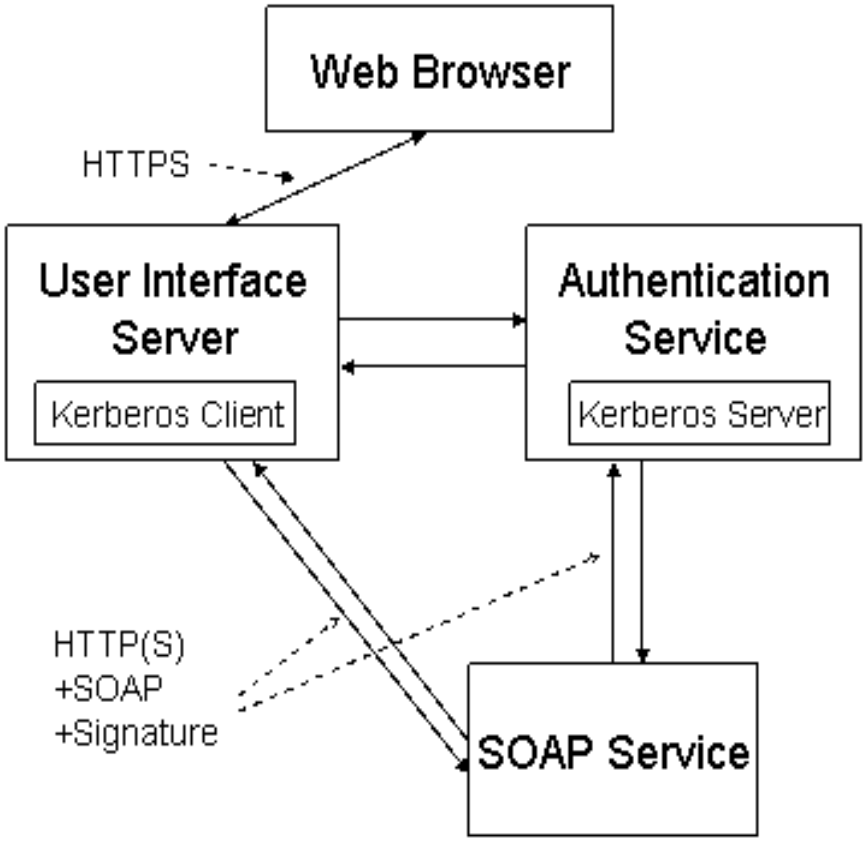

Figure 2 An assertion based authentication service using Kerberos.

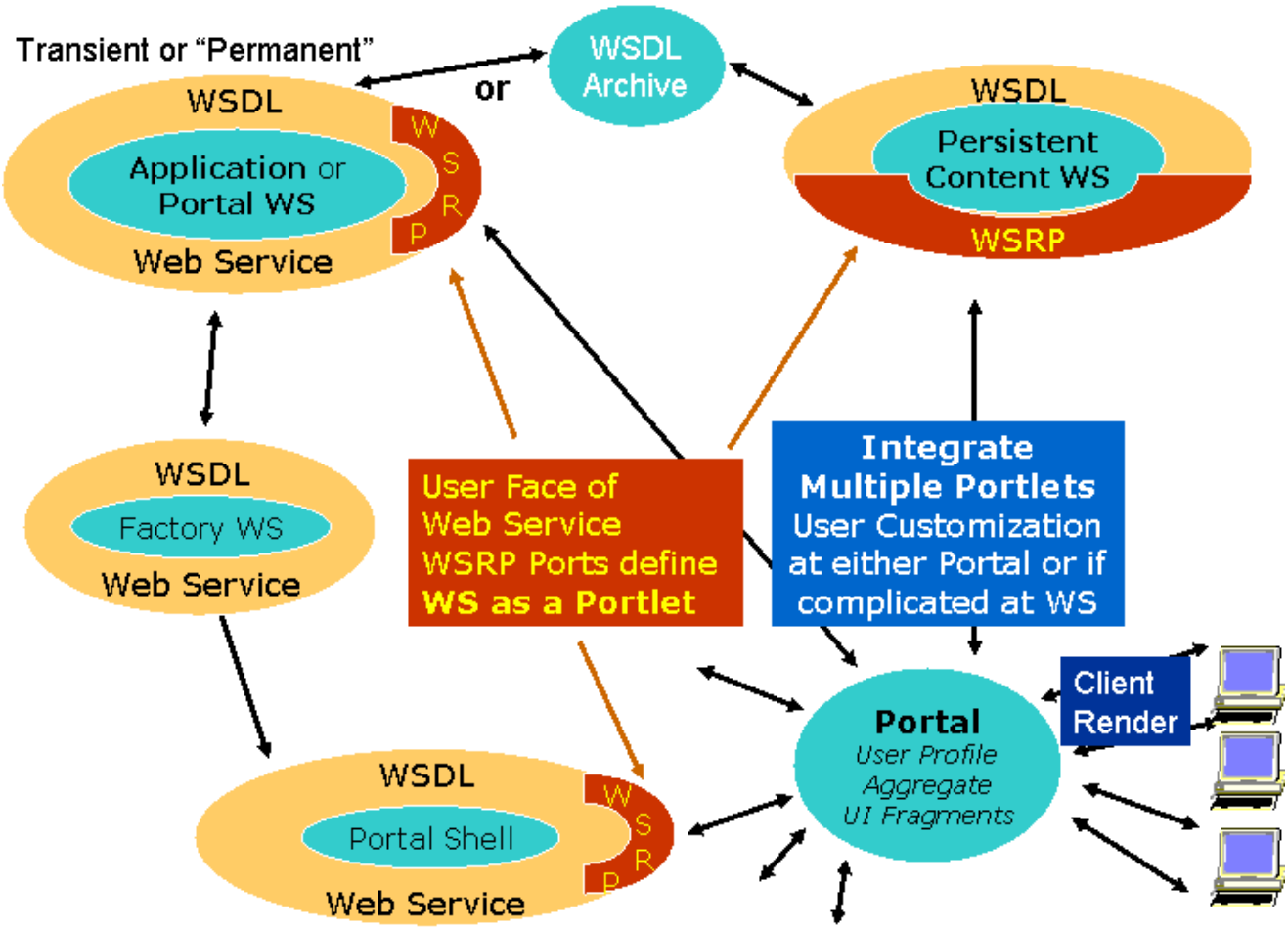

Figure 3 A schematic representation of the lifecycle of an application invoked through a portlet.