Toward an OpenSocial Life Science Gateway

Wenjun Wu[1], Michael E. Papka[1,2], Rick Stevens[1,2]
[1] Computation Institute, University of Chicago & Argonne National Laboratory, USA
[2] Mathematics and Computer Science Division, Argonne National Laboratory, USA

***Abstract*— To enable the Life Science community to fully use TeraGrid resources for computing and data management, we developed an integrated cyber computational environment named the Open Life Science Gateway (OLSGW) [1]. Based on a service-oriented framework, the gateway aggregates a group of bioinformatics applications and data collections into a web portal. Furthermore, this gateway provides a platform for life science researchers to collaboratively conduct bioinformatics computing, building their communities and sharing data and workflows.**

In the era of web 2.0, gadgets such as iGoogle gadgets are becoming increasingly popular for integration and customization of web contents from different sources. Social networking is also gaining attention because it has created powerful new ways to build virtual communities in a bottom-up manner. Google's OpenSocial framework [2] standardizes the practices of both gadget programming and online social networking, enabling web developers to write social gadgets that can run in any OpenSocial-compliant container. These new web technologies can leverage science gateway portals in terms of rich user interface and social network capability, which will promote the adoption of science gateways for advanced education purpose.

This paper introduces the OLSGW service-oriented framework and describes our efforts to develop OpenSocial gadgets for running bioinformatics analysis tools on the TeraGrid resources. The gadgets can be deployed in OpenSocial containers such as the iGoogle Sandbox [3] and Apache Shindig [4]. With these gadgets, biologists and biology students can easily run their analysis programs and browse outputs through social web sites.

***Index Terms*— OpenSocial, Gadget, Science Gateway, TeraGrid, Bioinformatics**

I. Introduction

Software tools and algorithms for high-throughput bioinformatics data analysis, such as sequence search, alignment, and protein structure analysis, are CPU-intensive, requiring tremendous computing power that is usually beyond the capability of clusters at a single biomedical research institute. Hence, such institutes often seek to take advantage of available high-end computing resources such as those of the TeraGrid to run their computing tasks on a much larger scale.

However, the complexity of Grid middleware makes it a challenge for biologists and bioinformaticians to use the TeraGrid resources without extensive knowledge of Grid technologies. To address this problem, we have developed a cyber computational environment named Open Life Science Gateway (OLSGW) that integrates a group of bioinformatics applications and data collections into a portal. Biologists with no prior experience of Grid computing can easily use this gateway environment to run their analysis programs and compose computational workflow scripts without the challenges of a sharp learning curve. This science gateway establishes a solid foundation for high-throughput genome and protein analysis workflows, helping scientists make great strides in solving the pressing problems faced by bioinformatics groups.

Like many science gateway systems, OLSGW follows the JSR-168 portlet specification by using the GridSphere [5] portlet container and customizing OGCE [6] portlets. Although the portlet solution is successful in integrating web contents for enterprise portals, it is not widely accepted in the public cyberspace, in which gadget based personal portals and social network sites have attracted millions of users, especially young people. To extend the OLSGW community and promote collaboration capability in the science gateway, we therefore started exploratory work in building gadgets for OLSGW services.

A standardized gadget framework is important for building AJAX gadgets for science gateway applications. Google's OpenSocial framework standardizes the practices of both gadget and social-networking sites, enabling web developers to write gadgets with social capability that can run in any OpenSocial compliant container. Moreover, an Apache open source project named Shindig was also launched last year to build a few reference implementations based on OpenSocial specification; and an OpenSocial developer community is emerging rapidly by following up the development of Shindig. Therefore it is the ideal platform for gateway developers to understand the internals of the OpenSocial framework and test their own gadgets.

This paper is organized as follows. In Section 2, we introduce our general framework for the open life science gateway. Section 3 presents the details about the design and implementation of gadgets in the OpenSocial framework. Section 4 displays a gadget example and points out the issues around gadget authorization. Finally, Section 5 summarizes our work and briefly outlines future studies.

II. General Services Framework for Life Science Researchers

The framework of Open Life Science Gateway is designed based on a service-oriented architecture: each bioinformatics command-line tool is regarded as a "Service", which can be described in a XML file in the community-specific format. The gateway generates necessary stubs from the XML description for a bio-application and deploys the stubs as an RPC service

in the system. Users are allowed to send requests to invoke the service through gadgets or a web services interface. On receiving user requests, the gateway creates service instances, and executes them on the computing and storage resources of the TeraGrid.

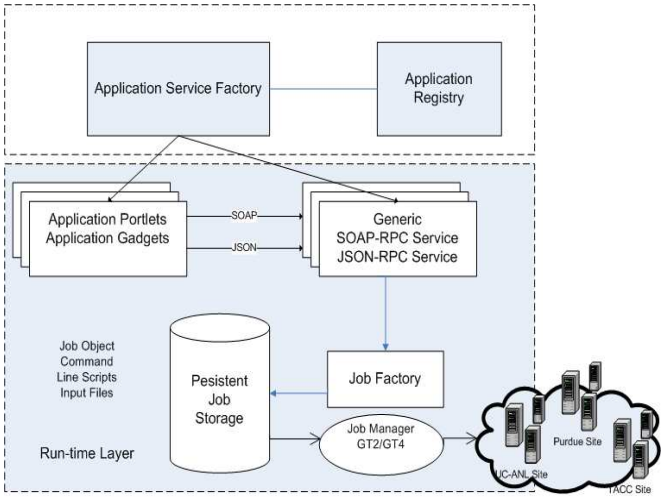

Figure 1 Open Life Science Gateway Framework

Figure 1 illustrates the primary components in this framework. Two layers are shown: the application service layer and the run-time layer. On the application service layer, the application registry keeps the meta-information of every legacy bio-application, which describes the command-line format, the installation locations, and the run-time requirement of the application. The PISE/Mobyle [7] package gives the command-line description for widely used bio-applications, which usually has argument lists, input files, and output files. Based on their description, bio-application RPC services can be provided for running the application on the TeraGrid resources. On the client side, from the same application description, the application service factory can build portlets and gadgets codes. The run-time layer supports the execution of deployed RPC services, including web services and JavaScript-oriented JSON-RPC services for creating, launching, and managing Grid jobs.

In this framework, a job is viewed as a running service instance. Deployed portlets, JSON-RPC services and web services call the generic Job Factory to create a job instance after they receive a user request, and deposit the job instance in a persistent job storage queue. The engine for job execution – Job manager – pulls the pending job instance from the job storage and submits it to the allocated TeraGrid sites through Globus GRAM.

A. Generic RPC Services

Generic RPC services, including both a generic SOAP-RPC service and a JSON-RPC service, unmarshal RPC requests from clients and create the job objects along with job execution scripts. This generic SOAP-RPC service serving the SOAP clients written in languages such as Java or Perl, is designed to handle SOAP requests for job operations including initiation, status query, result retrieval, and job destroy. It offers four major methods that can be invoked through SOAP.

TABLE 1
GENERIC RPC SERVICE INTERFACE

```
Interface Generic Web-Service
{
String JobID runJob(String application, Hashmap<String,String>
params);
String checkStatus(String jobId);
String getResults(String jobId);
String destoryJob(String jobId);
}
```

Currently, the generic SOAP-RPC is deployed in an AXIS container. Via SOAP::Lite library, Perl users only need to specify the parameters for a particular bioinformatics application and send a SOAP request to the generic SOAP-RPC service. Beside the SOAP-RPC service, the actual Java implementation of this interface can be exposed as a GridSphere portlet service and a JSON-RPC service.

The client stub of the SOAP-RPC service has been wrapped as a GridSphere portlet service, through which the portlets can call the SOAP-RPC service for submitting application jobs. In this way, the portlet container can run in a decoupled environment from the AXIS container and the Job Manager. If both can be hosted in a shared environment, it should be more efficient for the portlets to use local method calls instead of RPC.

The purpose of providing JSON-RPC directly to OLSGW gadgets is to implement a lightweight AJAX communication channel between the gadgets and OLSGW services. Normally it is believed that this approach should be faster than processing SOAP messaging in JavaScript because of the overhead caused by XML parsing. Just like the portlets, the JSON-RPC service can either run SOAP-RPC stubs to make a remote SOAP call or directly invoke the local method in the SOAP-RPC implementation. Since our JSON-RPC is Java based, it is also straightforward to expose the relevant Java classes through a JSON-bridge from the JSON-RPC-Java implementation [8]. Table 2 gives an example of calling the JSON-RPC service in JavaScript to run a clustalw application in OLSGW.

TABLE 2
UNITS FOR MAGNETIC PROPERTIES

```
function runClustalW(){
  allcookies = document.cookie;
  var ws_services_key = get_cookie("wskey");
  // create a JSONRpcClient object
  jsonrpc = new JSONRpcClient("/jobsubportlets/JSON-RPC");
  var params = new Object();
  // java class hint
  params.javaClass = 'java.util.Hashtable';
  params.map = {};
   params.map['actions'] = document.getElementById("action");
  params.map['quicktree'] = document.getElementById("quicktree");
```

```
params.map['outfile'] = 'OUTPUT';
 var seqinput = document.getElementById("seqinput");
params.map['infile'] = seqinput.value;
  // Hashtable params
  result = jsonrpc.JobService.run(ws_services_key, "clustalw", params);
```

B. Application Service and Job Factories

Two other components in OLSGW also help the generic RPC services to accomplish their tasks. One component is the Application Service Factory, responsible for parsing an application XML description, generating service stubs and formatting application-specific web pages and portlet codes. Using XSLT templates, the factory can create JSP pages, the portlet skeleton and gadgets that invoke the generic RPC services. The other component is the Job Factory. Based on the parsing output from the Application Service Factory and the command line arguments given by the Generic RPC service, the Job Factory builds a job execution script and constructs a job object in the persistence job storage. If the argument list contains input sequences, the Job Factory also copies the sequence data into a temporary file for data staging.

III. Building OpenSocial gadgets for OLSGW

Two types of OpenSocial gadgets exist: HTML gadgets and URL gadgets. We can develop either type for the Open Life Science gateway. Section 3.1 and Section 3.2 discuss the advantages and disadvantages of both approaches separately.

A. URL Gadgets

A URL gadget needs only a URL link pointing to the web resource that handles the gadget's user interface and programmatic logic. It works like a regular IFrame, which renders the content fetched from the remote site referenced by its URL. It is straightforward to wrap any HTML pages in the GridSphere portal of the OLSGW as URL gadgets.

A detailed guideline [9] is followed to turn these existing pages into gadgets. For OLSGW gateways, we write a login JSP page based on GridSphere's authentication mechanism and transform the portlet JSP pages for application job submission into HTML pages with JavaScript codes that call the generic JSON-RPC service. After being wrapped in a gadget description XML, each page is then turned into a URL gadget.

Although it is easy and quick to develop URL gadgets on the basis of your existing pages, there are disadvantages in this approach:

1. URL gadgets cannot make use of OpenSocial APIs to add more social-networking features and improve user experience in data sharing and workflow collaboration.
2. Some web browsers, such as Microsoft Internet Explorer and Apple Safari, do not permit third-party sites to set cookies according to their default security policy. Hence, OLSGW URL gadgets may encounter problems in these browsers. In particular, the cookies are not allowed to be created while users get authenticated in the OLSGW login gadget running inside an IFrame from the iGoogle sandbox. Other application gadgets cannot find the cookie to get the necessary user authorization information.

B. HTML Gadgets

With HTML gadgets, both issues can be addressed. A HTML gadget has all the logics in its body, which usually consists of the gadget XML as well as HTML markups and JavaScript codes. The OpenSocial specification also enables gadgets to aggregate information from multiple sources and interacts with existing services.

As presented in Section 2, OLSGW gadgets need to communicate with a hosted JSON-RPC service to run bioapplications. Normally a JavaScript library implements JSON-RPC by directly making an AJAX request. However, browser security models prohibit gadget JavaScript from making cross-domain AJAX requests, which prevents developers from using standard AJAX libraries to fetch content from other sites. Fortunately, the Gadgets API provides a function called makeRequest that allows the gadget JavaScript to communicate with remote services through a proxy in the gadget's OpenSocial container.

In OLSGW, the JSON-RPC-Java package is used to transparently call server-side Java code from JavaScript. It includes a JavaScript class, named JSONRpcClient that dynamically queries the object RPC methods available on the RPC server and creates JavaScript function stubs for each method. A function stub makes a XMLHttpRequest call to pass arguments to the remote RPC server, get a result JSON object back, and return it to the user's callback function.

Obviously it is essential to replace the XMLHttpRequest calls in the function stubs by gadgets.io.makeRequest if we want gadgets to use the JSONRpcClient for JSON-RPC calls. Moreover, major refactoring must be made in the JSONRpcClient constructor because it carries out the query of the available methods through a synchronous XMLHttpRequest. Because the gadgets.io.makeRequest can execute only asynchronously, a callback handler should be added in this constructor to handle the initialization of function stubs and be guaranteed to be completed before subsequent JSON-RPC calls.

C. Gadgets Authorization

Using HTML gadgets also makes it possible to set up a new way of authentication and authorization for accessing remote RPC services. As we pointed out in Section 3.1, URL gadgets can use cookies only for tracking remote login and authorization information. We need a common user authorization delegation mechanism to allow users to directly visit the authorization page from the science gateway site and authorize the OpenSocial container to call the RPC services. One solution is a new security protocol, named OAuth [9], that enables developers to offer their services to gadgets running outside their service containers, without forcing their users to expose their passwords and other credentials to gadget host environments.

OAuth authentication is done in three steps:

1. The consumer obtains an unauthorized request token.
2. The user authorizes the request token.
3. The consumer exchanges the request token for an access token.

In the context of OLSGW, the OAuth consumer is referred to the makeRequest proxy in the OpenSocial container. The JSON-RPC service provider plays the role of the OAuth provider. By introducing a few OAuth servlets and a filter chain for the JSON-RPC service, we can implement the OAuth for authorizing the JSON-RPC service.

Three parties get involved in the OAuth procedure: gadgets, the OpenSocial container, and the JSON-RPC provider. Inside the OpenSocial container, a makeRequest proxy handles the OAuth for remote data fetching, and a callback servlet responses to the callback requests generated in the authorization step. On the side of JSON-RPC provider, there are three URLs for obtaining a request token, authorization approval, and an access token, which can be mapped to three different servlets or filters. An extra security filter is also needed to check the OAuth information to make sure that the access token is valid to invoke the JSON-RPC service. During an OAuth session, the JSON-RPC provider has to track the life cycle of the request token, the access token, the associated token owner (user ID), and the JSON-RPC session.

Figure 2 depicts the message flow among the three parties through the whole OAuth protocol procedure.

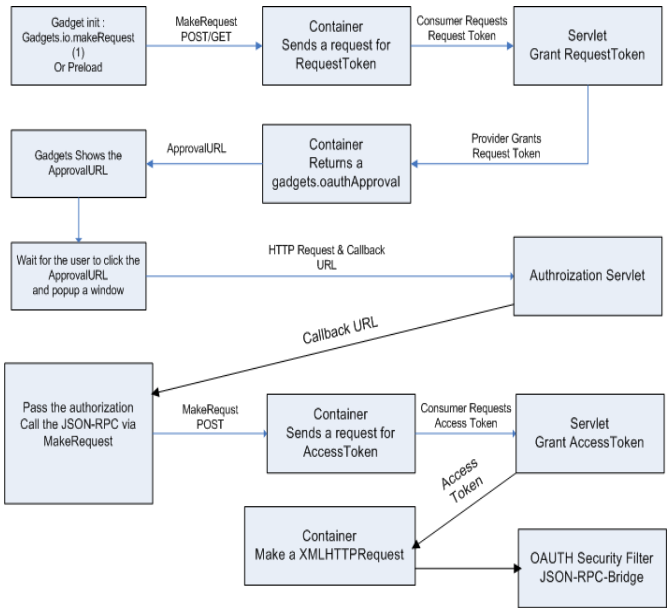

Figure 2. OAuth implementation for JSON-RPC

After the gadget initiates a gadgets.io.makeRequest with the relevant OAuth attributes, the container sends a message to the OAuth provider (request_token_servlet) to obtain a request token. When the unauthorized request token is passed back to the container, it creates an oauthApprovalURL that is the concatenation of the OAuth authorization URL and the callbackURL. The oauthApprovalURL link is presented to the user in the gadget. After the link is clicked, a popup window is displayed for authorization. The authorization form is listed as follows:

TABLE 3
AUTHORIZATION FORM

```
<form name="authZForm" action="authorize" method="POST">
      <input type="text" name="userId" value="" size="20" /><br>
      <input type=”password” name=”password” /><br>
      <input type="hidden" name="oauth_token" value="<%= token
%>"/>
      <input type="hidden" name="oauth_callback" value="<%=
callback %>"/>
      <input type="submit" name="Authorize" value="Authorize"/>
 </form>
```

The form asks the authorization servlet in the JSON-RPC provider to check the user name, the password, and the request token and decides whether to grant the request from the user. Our authorization servlet relies upon GridSphere's authentication mechanism to validate user's credential and redirects the popup authorization page to the callback URL.This callbackURL refers to the callback servlet running on the OpenSocial container, which simply closes the popup window.

As soon as the popup window is closed, the gadget will automatically call the JSON-RPC service to list all the available method. The makeRequest proxy firstly requests an access token to the OAuth provider (access_token_servlet); and after the arrival of the granted access token, it posts the XMLHttpRequest to the JSON-RPC URL. Carrying the valid OAuth tokens, this post request passes through the OAuth security filter and reaches the JSON-RPC servlet at the end of this filter chain. When the proxy gets the reply from the servlet, it forwards the reply to the gadget, in which the list of the available RPC methods can be found. It enables the gadget to make the subsequent RPC calls to run the bioapplications.

IV. IMPLEMENTATION AND EXPERIENCES

Adapted from the existing application portlets, three application gadgets – BLAST, InterProScan, and ClustalW – have been built for the Open Life Science gateway. We have tested them in the OpenSocial containers such as Shindig and iGoogle Sandbox. Figure 3 and 4 show the portlet page and the gadget page for the clustalw application separately.

Although OAuth provides a standardized way to achieve authorization delegation for gadgets, there are still a lot of issues in this specification, especially on the management of OAuth tokens, customer keys, and secrets. One of the problems is how to automatically refresh access tokens while gadgets may want to keep sending multiple RPC requests without repeating the authorization procedure. By the OAuth proxy [11], if a gadget has passed through the OAuth procedure and obtained an access token, Shindig and the iGoogle Sandbox can renew the token for the gadget for subsequent io.makeRequests. But when a user has multiple gadgets that actually access the RPC services hosted in the

same container, he still has to do separate OAuth authorizations for running them because the OAuth proxy distinguishes different gadgets by their URLs and manages their tokens separately. One of possible workarounds is to add a group token management on the side of JSON-RPC provider so that it can generate an access token for a gadget without user authorization if the gadget owner already did so for another gadget in the same group. And through the pub/sub feature in the OpenSocial API, the gadgets in a group can notify each other whether they have finished authorization.

Another problem concerns the management of OAuth customer keys and secrets in both the consumer side (OpenSocial container) and the provider side. It is not defined in the OAuth specification but is left to container developers to decide how to implement the OAuth store of the keys and secrets. As a reference implementation, Shindig just puts the store in a flat file and wraps in one of its jar files. For the iGoogle Sandbox, one must send a email to oauthproxyreg@google.com with gadget URL, OAuth consumer key, and secret to register a shared secret. On the JSON-RPC side, currently we use only a single file for the storage of the customer key and the secret because only a few application gadgets are needed for OLSGW.

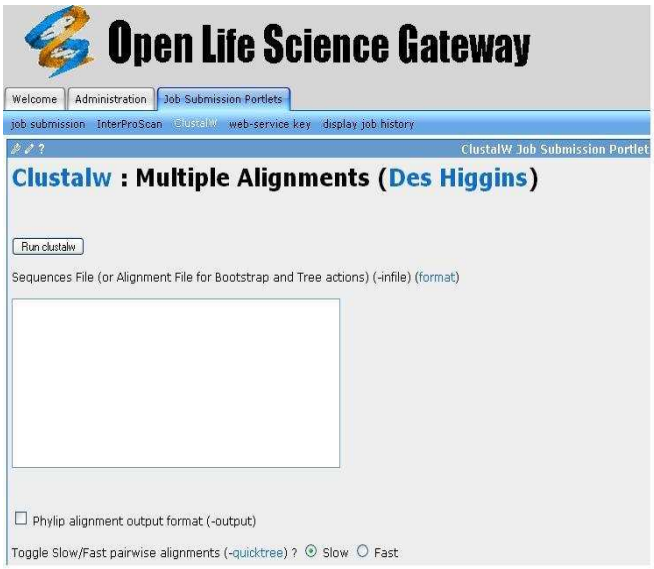

Figure 3 Clustalw Job Submission Gadget

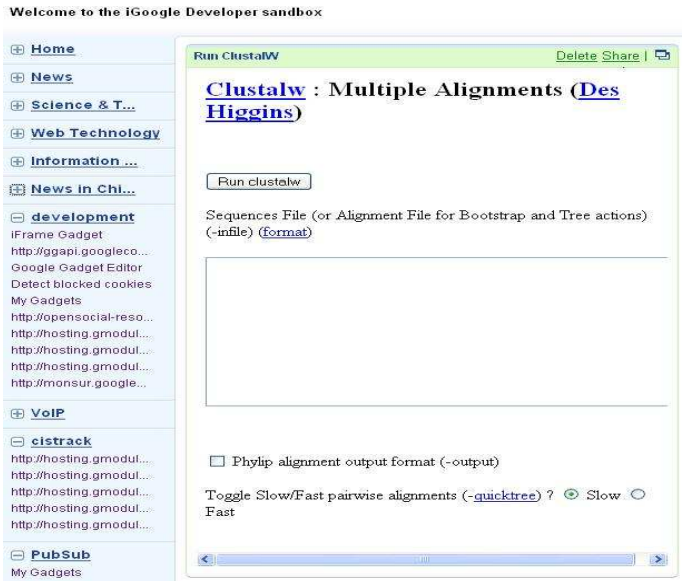

Figure 4 Clustalw Job Submission Gadget

V. Conclusion and Future Work

The Open Life Science Gateway provides services to the life science community, enabling easy access to the TeraGrid resources for computing and data management. The extensible and service-oriented framework is introduced in developing the OLSGW to achieve the integration of many command-line bioapplication tools. To promote the further application of the OLSGW, especially as an educational science portal, a few open social gadgets are developed to allow users to run bioinformatics analyses through commercial OpenSocial sites such as iGoogle Sandbox. The paper presents our experience in building OpenSocial applications based on the existing science gateways hosted as GridSphere portlets in an Apache Tomcat container. The paper discusses such problems as how to call JSON-RPC services from OpenSocial gadgets and how to implement authentication and authorization for the remote JSON-RPC service.

Since the OpenSocial framework is still at an early age, a lot of issues still remain. For example, it is not clear how to support a single authorization for a user to access a group of gadgets belonging to the same service. Moreover, we have to test our gadgets in other OpenSocial complaint containers such as MySpace and Hi5.

Acknowledgment

This work was supported in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357 and the National Science Foundation by grant OCI-0504086.

References

[1] Wenjun Wu, Rob Edwards et al, TeraGrid Open Life Science Gateway, TeraGrid 2008 conference, June 9-13, 2008, Las Vegas,.
[2] OpenSocial Specification, http://www.opensocial.org/

[3] IGoogle sandbox, http://code.google.com/apis/igoogle/docs/igoogledevguide.html
[4] Shindig, http://incubator.apache.org/shindig/
[5] GridSphere, www.gridsphere.org
[6] Open Grid Computing Environment, http://www.collab-ogce.org/ogce/index.php/Main_Page
[7] C. Letondal, "A Web Interface Generator for Molecular Biology Programs in Unix,"
[8] Bioinformatics, 17(1), pp 73-82, 2001
[9] JSON-RPC-Java, http://jabsorb.org/
[10] Google Gadgets Page, http://code.google.com/apis/gadgets/docs/fundamentals.html
[11] OAuth, http://oauth.net/core/1.0
[12] OAuth Proxy, https://sites.google.com/site/oauthgoog/oauth-proxy