2-18-2010

Cyberaide JavaScript: A Web Application Development Framework for Cyberinfrastructure

Fugang Wang

Cyberaide JavaScript: A Web Application Development Framework for Cyberinfrastructure

Fugang Wang
Service Oriented Cyberinfrastruture Lab, Rochester Institute of Technology
Bldg 74, Lomb Memorial Drive, Rochester, NY 14623-5608
Email: fxw7441@rit.edu

February 18, 2010

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Science

Committee Members:

Chair: _____
Dr. Hans-Peter Bischof
Associate Professor
Department of Computer Science, Rochester Institue of Technology
Rochester, NY 14623

Reader: _____
Dr. Gregor von Laszewski
Assistant Director of Cloud Computing
Pervasive Technology Institutue, Indiana University
Bloomington, IN 47404

Observer: _____
Dr. Minseok Kwon
Assistant Professor
Department of Computer Science, Rochester Institue of Technology
Rochester, NY 14623

Contents

List of Figures

List of Tables

Abstract

This thesis work introduces a service oriented architecture based Grid abstraction framework that allows users to access Grid infrastructure through JavaScript. Such a framework integrates well with other Web 2.0 technologies since it provides JavaScript toolkit to build web applications. The framework consists of two essential parts. A client Application Programming Interface(API) to access the Grid via JavaScript and a full service stack in server side through which the Grid access is channeled. The framework uses commodity Web service standards and provides extended functionality such as asynchronous task management, file transfer, etc. The availability of this framework simplifies not only the development of new services, but also the development of advanced client side Grid applications that can be accessed through Web browsers. The effectiveness of the framework is demonstrated by providing an Grid portal example that integrates a variety of useful services to be accessed through a JavaScript enabled client desktop via a Web browser, as well as the opensocial gadgets for Grid task management and file transfer. Overall, Grid developers will have another tool at their disposal that projects a simpler way to distribute and maintain cyberinfrastructure related software, while simultaneously delivering advanced interfaces and integrating social services for the scientific community.

Keywords: Cyberinfrastructure, Web Service, Web Application, JavaScript, Grid Computing, Grid Portal.

1 Introduction

We are relying on high-performance computing more than ever to do scientific research. From the traditional high energy physics to Gene research, or from drug discovery to aerodynamics work, we cannot perform the works otherwise due to the system complicity requires more computing power and/or produces huge mount of data to be processed. Instead of maintaining such computing facil-

ities independently by each institute and organization, Cyberinfrastructure [1] proposes to build a new environment that could efficiently connect those computing facilities to achieve a higher goal for scientific discoveries. Teragrid [2]is a result of such effort, which connects computing facilities from institutes across the nation to provide huge computing and data processing potential to scientific users. Grid computing is usually the underlying technology to achieve this. However, setting up a Grid computing environment to use the cyberinfrastructure is a pretty complicated work that requires skills not possessed by all the users from different fields.

Grid abstraction has been proved through the Java CoG Kit [3] project an ideal way to lower the entry barrier of the Grid computing. To further continue this approach, together with the booming Web 2.0 and Software as a Service(SaaS) trends, we consider to construct yet a high level abstraction that enables Grid access from a web browser client. The thesis work provides a Service Oriented Architecture (SOA) [4] based Grid abstraction framework that allows us to access Grids through web client using JavaScript. In the framework, we are enhancing the previous approach with a number of advanced services as well as targeting JavaScript as the language of choice for the client to support Web 2.0 style portals. Through the framework we can integrate with a variety of Grid middleware and obtain access to the Grid fabric. The work focus primarily on the integration with Globus [5] and access to the TeraGrid. The framework contains a useful set of JavaScript toolkit to simplify this access.

One of the advantages of the JavaScript API is that we can integrate a large number of commodity libraries available in JavaScript to go beyond the traditional use of Grid technologies. Hence, we are able to leverage from data-structure libraries, social networking and communication libraries to enable Web 2.0 programming features and allow access to additional commodity cyberinfrastructure that would otherwise be difficult to be achieved. As a result the framework will be more than just a Grid client library. In order to emphasize this difference and its ability to function as an aide for integrating cyberinfrastructure in general, we use the term *Cyberaide Javascript* as the name of the project.

The article is structured as following. We will discuss the background and the related research, technologies and standards first. It is followed by the system design section, which covers the architecture, service stack design and client side JavaScript library design. In the implementation section, the service stack in server side and the client side library are implemented, with some technical details covered. Evaluation & use cases section introduces several examples of web applications for cyberinfrastructure developed based on the framework and the JavaScript API. A list of deliverables from this project and some important documentations generated are listed as appendices.

2 Background and Related Research

Harnessing super computing power and processing large amount of data to get useful information is not always an easy problem. To better resolve this issue, the NSF answered with the concept of Cyberinfrastructure [1], the main goal of which is to lower the existing entry barriers to the high performance computing. The work of this thesis is part of the Cyberaide (Cyberinfrastructure Agile Development Environment) project [6]. The goal of this project is to develop methods and tools to address the development, deployment and use of the advanced cyberinfrastructure. Figure 1 displays the big picture.

As an essential part of the Cyberaide project, Cyberaide JavaScript is to develop a framework that deals with Web application development for the advanced cyberinfrastructure. In this section we will introduce and discuss the background on which this thesis work bases as well as the related research & technologies.

2.1 Grid Computing

Grid computing [7] is a metaphor that implies the access to computing power is as easy as accessing an electric power grid. Practically it is a kind of distributed computing, containing a number

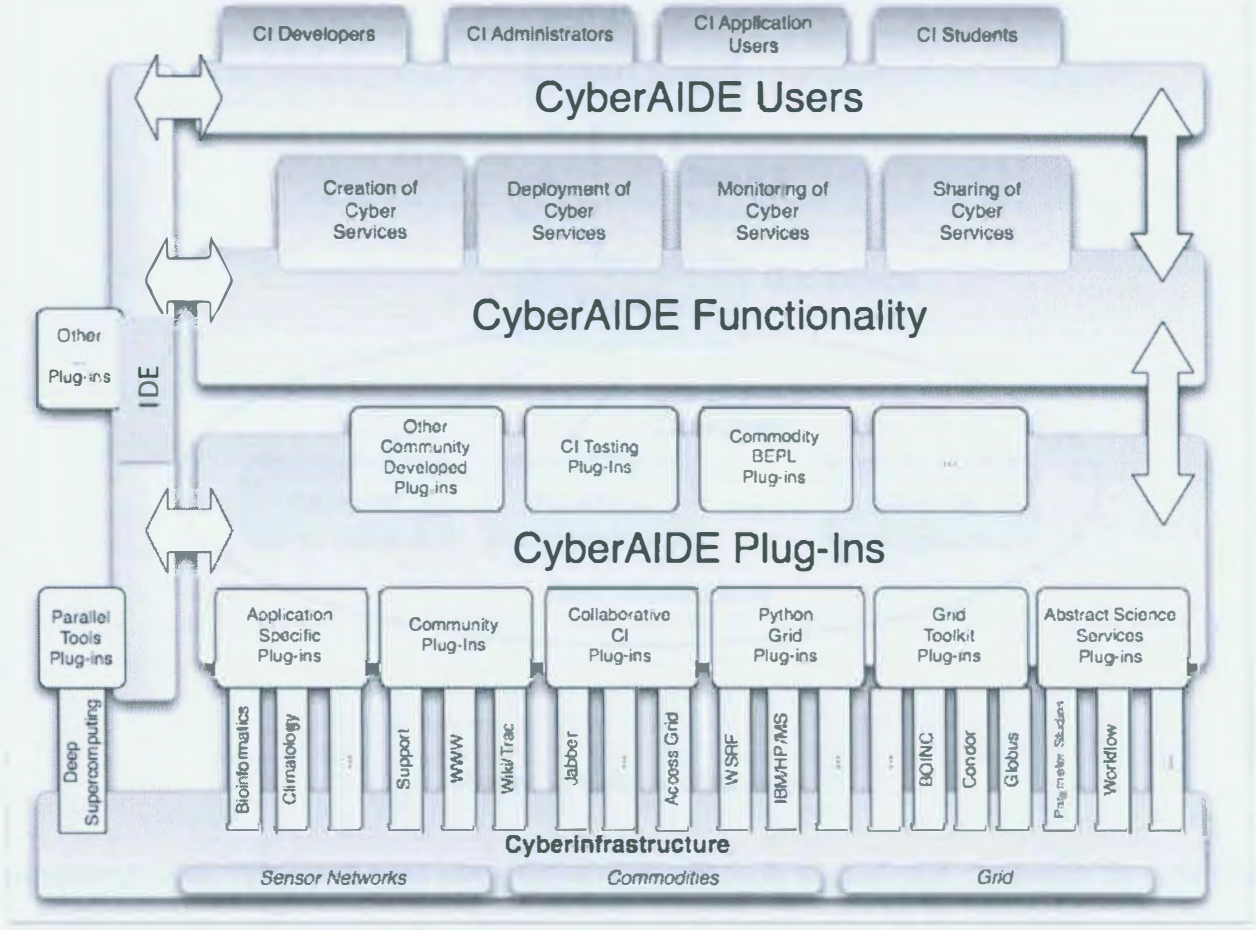


Figure 1: Cyberaide Framework

of geographically distributed computers connected by most likely high-speed networks under different administration. The loosely coupled computers may share computing resources like CPU cycles, storage spaces, etc. Grid provides an alternative way to access high-performance computing using a number of available computing resources, including a range from fairly cheap to expensive mainframes or supercomputers. Currently the Teragrid [2] is the nation's largest collaborative effort providing an advanced cyberinfrastructure based on Grid technologies.

A simplified Grid system is illustrated as in Figure 2. It is based on the *job submission, status query, result retrieval* paradigm. Grid clients interact with some Frontend in the Grid, and the Frontend will do the job dispatch and coordination between the Backend, which does the actual

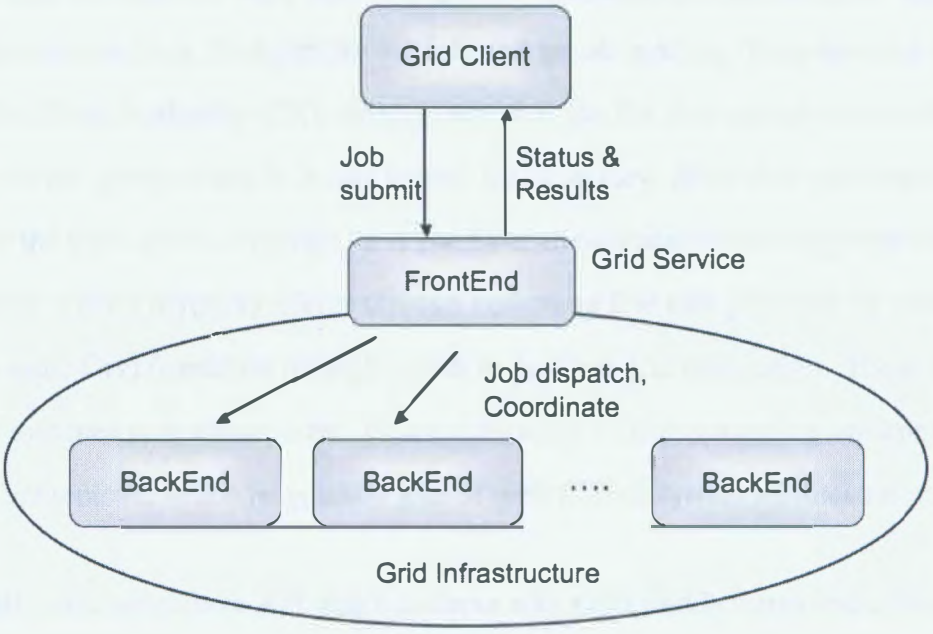


Figure 2: Grid Architecture

computing work. This will need some framework and tools to deal with the issues aroused like security, resources discovery and monitor, etc.

Globus Toolkit is the de facto standard Grid computing middleware. It is the effort from an open source project. Globus provides ample functionalities for underlying job dispatch, and elementary shell based interface. Grid users, after authenticated himself/herself successfully, could submit a job to execute, query its status, and transfer large mount of data, etc. Globus utilizes internally Service Oriented Architecture. Together they are called Open Grid Service Architecture (OGSA). Globus Toolkit uses the Grid Security Infrastructure (GSI) as the security infrastructure, which provides transport layer security and message level security. A proxy certificate store is often used to deposit certificates and to facilitate the certificate credential delegation, by which some agent can delegate the execution of a complete workflow on behalf of the user. To use the toolkit in Grid

environment, first one need to setup and configure a Globus client environment, which includes some basic modules such as the myproxy module and gsissh module. Then we need to configure the trusted Certificate Authority (CA), making sure that the CA that issued the certificate for the Grid system we are going to use is in our trusted CA directory. After that you need to request a certificate that the Grid system will trust, or if you have already obtained the appropriate certificate, you can retrieve it from myproxy server through command line tool provided by the toolkit, and then login to some Grid frontnode through gsissh to perform grid computing. These steps are not trivial, and sometimes very error-prone. This makes setup a Grid computing environment is very complicated and tedious, which is typically jobs of well-trained system administrators.

Java CoG Kit is a set of Java API that interfaces with Grid middlewares and commodity technologies. It provides a higher level abstraction of Grid services, through which grid users can get a user-friendly and more convenient environment to access Grid services that the Globus toolkit provides. The abstraction and provided API also simplify Grid developer's work, but do not expose every feature of the Globus toolkit. Java CoG Kit introduces a full-fledged workflow description language and engine called Karajan. The Karajan workflow engine provides an extended XML description in order to to describe workflows. This makes it easy to construct complete workflows, through use of dependencies and parallel constructs. Earlier version of the Java CoG Kit even provides a GUI for user to construct workflow. A Grid user could construct and edit a workflow by drag and drop manner. Due to rich features and easy usage, Java CoG Kit is widely used as the mechanism to interact with underlying Grid middleware and Grid infrastructure.

2.2 SOA and Web Service

Service Oriented Architecture (SOA) is a new approach to construct distributed application, especially business application [8, 9].

It is also widely used in Grid community including Globus Toolkit since version 4. The sim-

plest SOA based use case is a 2-tier architecture, one side providing service and another side consuming the service. While for some other cases, some mediator is needed between the service requestor and service provider to gain the interoperability. In this case, the mediator is both a service provider and service requestor.

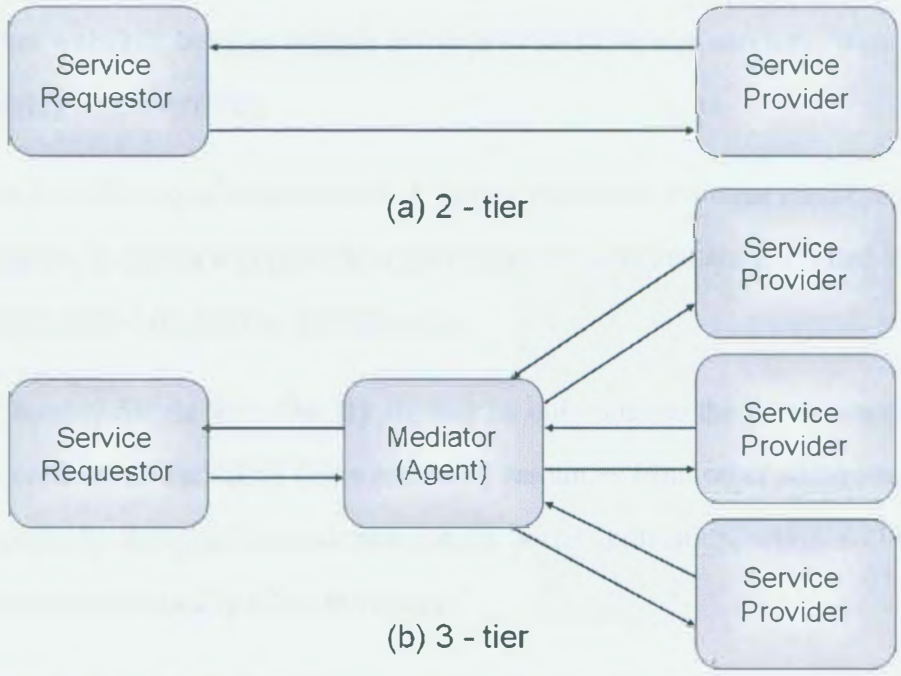


Figure 3: SOA

The use of SOA can make the entities loosely coupled, so that each part can be modified and updated independently without impact on other parts, as long as the service contract is kept. Web service is a specific form of Service Oriented Architecture, which constructs a SOA through Web. So it can be widely used since web clients can be found almost in every computer for now. A web service is defined using Web Service Description Language(WSDL) [10]; to discover a web service, Universal Description, Discovery, and Integration (UDDI) [11] specification is used. Many web service standards and specifications are still emerging as well as evolving now, while some address security, reliable message transfer, and others transaction. However, the most commonly

used underlying protocol is SOAP [12], using XML to transfer messages over HTTP.

2.3 Web Portal & Grid Portal

Traditionally a web portal is a website that is considered as an entry point to other websites. In these days since web services technologies are booming, the concept should go beyond the concept of entrance to other websites, but also include entrance to multiple web services. Web portals usually have the following characteristics:

- Multiple sites/functionalities/services. A portal introduces a unique entrance to these multiple resources, in such a way provides convenient for users that are interested in certain group of functionalities provided by multiple sites.

- Single entrance/Single Sign-On. By signing on only once to the portal, users don't need to provide credentials each time when accessing resources from other administrative domains. A user typically delegate the credentials to the portal application, which will be authenticating the user automatically when necessary.

- Consistent look and feel. Providing a user friendly interface and consistent look and feel can increase users' experience.

- Personalization. This is helpful when the target users of the portal may have various requirements and preferences.

Obviously these characteristics define the merits why we are using portal.

Grid portal shares the same features as typical portals. It provides a uniform environment for Grid users to interact with Grid services. Some basic functionalities include authentication, submitting jobs and monitoring jobs execution status. Other functionalities supported could be client side workflow composition, Grid information services, and collaborative environment. Specially

we need to pay more attention on user authentication and authorization since Grid is typically a system that across administrative domains.

2.3.1 Web Portal Standards and Technologies

In this section we discuss the traditional web portal standards and technologies, to be specific, JSR168 and WSRP.

In the very early web era, a web portal was usually constructed by grouping the links to different sites into one portal page. With the increasing adoption of Java in multi tier web application development, A Java Portlet Definition Standard (JSR168) [13] was formulated. The standard follows the scheme of Java servlet, but introducing portal specific standard features.

JSR 168 defines set of concepts like

- Portal
- Portlet
- Portlets container

which has relationships depicted through a portal structure example as in Figure 4.

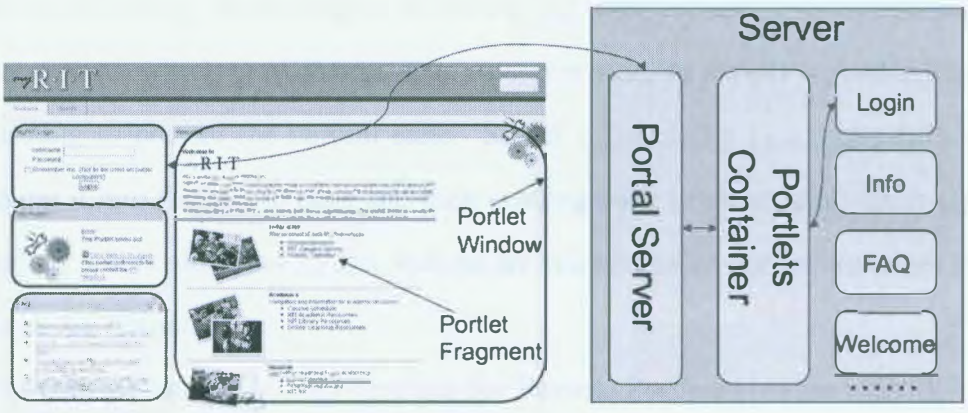


Figure 4: A portal structue example

Portlet mode could be one of

- View/Edit/Help
- Custom

Portlet request handling defines types to deal with different request

- processAction
- render

Portlet request could be respectively

- ActionRequest
- RenderRequest

And so does the Portlet Response

- ActionResponse
- RenderResponse

It works as following. According to its own mode, user request from a web page could be some action request, which may change state in server side, or simply a render request, which is just requesting to present the current state. Server responds by generating HTML fragment and assembling it into the whole page and then sending back to client side. Each portlet has its separated area in client side, and all the portlets are resides inside portlet container that controls the behavior of the portlets.

Another existing standard is Web Services for Remote Portlets [14] ver1 (WSRP v1), which defines set of web services interfaces to interact with remote portlets. This standard, if being followed, usually works together with JSR168.

JSR168 has a reference implementation called Apache Pluto [15], which could be used as a portlet container. Developers could develop JSR168 compatible portlets and deploy them into Pluto. Similarly WSRP also has a reference implementation Apache WSRP4J [16]

Other popular portal frameworks include uPortal [17], OpenPortal [18], etc.

The above mentioned are mostly based on relatively old web standard and technologies. Some emerging standards include JSR286, which is version 2.0 of the Java portlet specification, and WSRP ver2, would obsolete the current JSR168 and WSRP v1. They claims to support Web2.0 features like Ajax, REST. JSR286 would deal with inter-portlet interaction, which would be beneficial for flexible data aggregation and presentation which are merits of Web2.0 based portals.

2.3.2 Web2.0 technologies and implication to portal development

The Figure 5 and Figure 6 shows examples of portal that based on Web2.0 technologies.

Figure 5: An Example Ajax Portal

They have similar structure and functionalities as the old ones, in users' perspective. But it

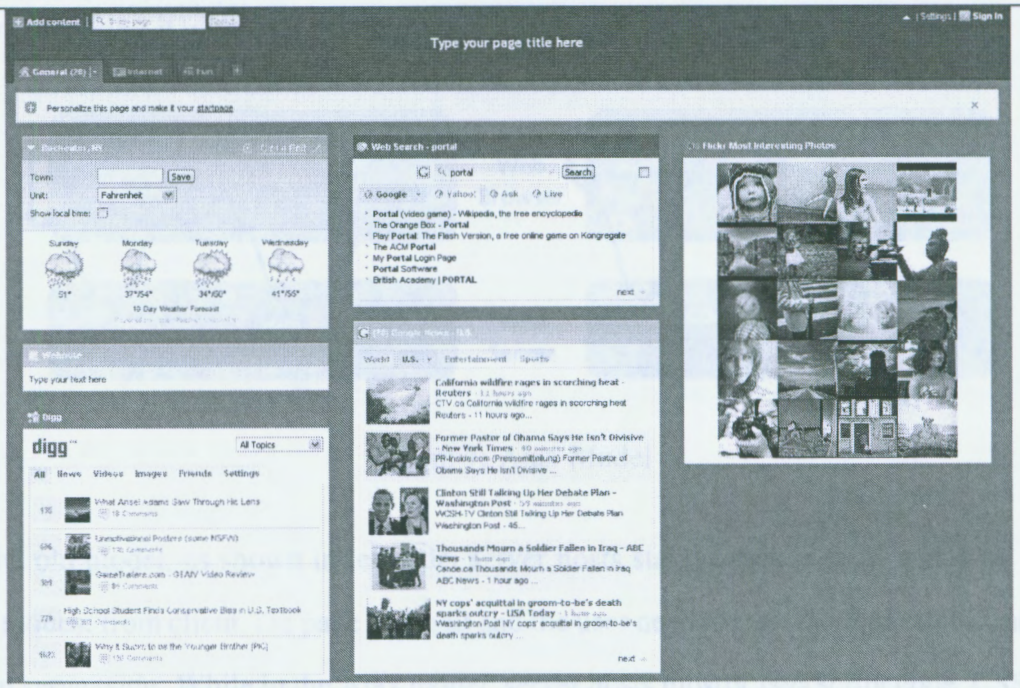


Figure 6: Netvibe Ajax Portal

leverages merits from web2.0 technologies and design approaches and thus has advantages like more flexibility to aggregate and present info, increasing user experience by providing more user friendly interface and decrease user waiting time by using Ajax model.

Web2.0 is still a very buzz word in its meaning, even after several years when it was coined. Basically it's a set of technologies and design patterns followed by developers, which includes but not limited to

AJAX Asynchronous JavaScript and XML (Ajax) Representational State Transfer (REST) [19] Atom [20] and RSS [21] Mashup

Ajax itself is also a set of design patterns, which usually includes features like XML [22] as data interchange medium. Asynchronous communication with server side. XHTML [23], CSS [24], and JavaScript [25] to manipulate and render DOM components.

By following Ajax in development, it decreases user waiting time and increase user experience, which makes web applications more like desktop applications. Figure 7 depicts the merits of this

model by comparing it with the conventional web interactions.

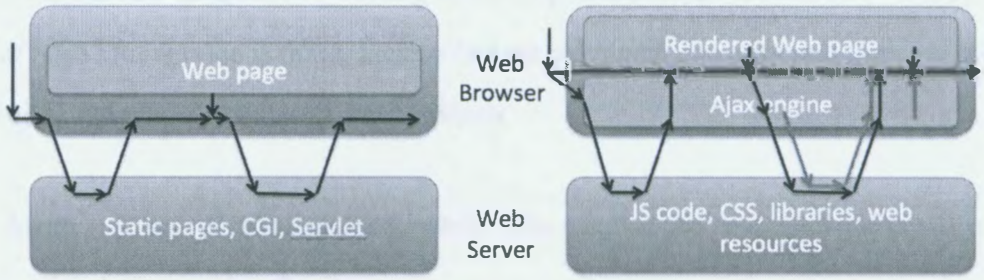


Figure 7: Ajax Model

For the old model, as shown in left side, server hosts static pages mostly, and each time user initiate a request from client, the page will halt and the user need to wait the requested content come back from server side. While in the Ajax model, server hosts mostly JavaScript code, CSS files and other resources. During user interaction, the JavaScript code, through Ajax engine in web browser, would interact with the server side to get back the necessary data to present, typically in XML format. By adding another layer, JavaScript code could retrieve data while not blocking current user's activities, and the result retrieved would be reflected in client side by callback function in JavaScript. It also has feature like secretly prefetch data that most likely would be used in the next steps. Thus when user performs a new request later, the result could be displayed instantly. A good example for this is Google Maps. When you focus on some place and display a certain area in window, the data of adjacent areas are prefetched during this time. So when the user pans the map zones, typically it has a very fast response time.

REST The term REST comes from Roy Fieldings doctoral dissertation [19] in 2000. It defines resource as functionality with state. A web service is called RESTful when it follows REST design approach. RESTful service has advantages of simplicity and requiring less processing and thus lower overhead, comparing with SOAP [26] based web services. Each resource is unique addressable through URI. It simply uses HTTP verb GET, PUT, etc. to operate on the resources

defined. It eliminates the use of additional message layer such as XML. Since state is implicitly contained in resources, session maintaining using http cookies [27] could be avoided. However a drawback of RESTful service is that it lacks a full set of standard to augment the web services such as WS-Security [28] in SOAP based web services.

RSS and Atom A server publish feed, while client subscribe and read them. It is a publish subscribe model, but provides flexible aggregation and presentation mechanisms by applying a standard format for the feeds.

Mashup Mashup is an approach that mingling and combining data from multiple sources, and presenting them in a new perspective. A typical example is online gas station locator while combining address data of gas stations from a certain area and map info from such as Google. By combining data from the two sources, it presents them in a new form. Mashup typically provides more meaningful and more useful information by combining and sometimes derives new information through that.

Web2.0 technologies implication on portal development As a comparison, we list the characteristics of the prevailing JSR168 approach and web2.0 approach for portal. JSR168 Portal

- Generate markup segments for each portlet and assembly them into a full page.
- Info aggregated at server side.
- Portlets displayed side-by-side.
- Old technologies and about to be outdated.

Web 2.0 Portal

- Combine raw data from different site/web services.

- Content aggregation can happen in server side or client side.
- Could combine data from different source and present in a totally new way.
- Web 2.0 technologies used to provide better user experience.

Through the comparison it is obvious that web2.0 is good to apply to portal development.

2.3.3 Grid portal - a survey and comparison

Grid portal development has been going on for quite some time. The first usable library to support the development of Grid portals was the Java CoG Kit [29]. Based on the premise of the Java write-once-run-everywhere concept the Java CoG Kit was designed explicitly to be a 100% Java-based library. Originally, Applets were developed that were soon enhanced by JSR168 [30] compatible portlets. These portlets were then integrated and enhanced as part of the Open Grid Computing Environments (OGCE) Project [31] and by Gridsphere [32]. Together these three projects (CoG Kit, OGCE, and Gridsphere) build a major foundation for the TeraGrid portal [33], which is one of the premier NSF sponsored resource in the U.S. to obtain access to Grid resources. TeraGrid uses the Globus Toolkit [5] to manage its Grid resources.

Gridsphere portal framework

- 100% JSR168 compliant
- Easy to develop and integrate new porlets
- Easy to create customized portal layouts
- Built-in support for Role Based Access Control
- Data persistence using Hibernate
- Localization

- Open source

The Open Grid Computing Environments Portal and Gateway Toolkit (OGCE Portal)

- JSR168 compatible portlets
- Grid credential management
- Secure remote file management and code execution
- Views of Grid Information service (GPIR)
- Workflow composers (XBaya)
- Open source

Teragrid user portal

- Provides Teragrid related information.
- GSI-SSH terminal to login resources through inside a web page.
- Uses GridSphere for some functionalities like file management.

Other portals also exist, like Genius, Legion, Pegasus, etc.

However, the current generation of Web based technologies have integrated JavaScript as one of the major offerings. Unfortunately, to date no JavaScript framework exists that lets us access the Grid easily. This is the main motivation of the work conducted through this thesis.

2.3.4 Approaches to access Grid

In this section we will summaries the approaches to access Grid.

First we have tools provided by Grid infrastructure. Some of them provides specific tools like job queue management, while others provides a full infrastructure to construct and manage a Grid.

This approach is most likely platform dependent, and requires a lot time to setup and configure the system. Some examples of this category are:

- Condor [34], PBS, LSF, SGE
- Globus [5], UNICORE [35], EGEE, Legion

Middleware/Upperware is built upon the Grid infrastructure, wrapping the functionalities and provides an abstraction layer for user. This approach provides ability to across platform and even across different Grid infrastructrre. It also requires less work to setup and configure. Some typical examples are:

- CoG Kit [36]. On top of Globus, while some components are used by Globus, it provides command line tools and Java library to access Grid. The built-in Karajan workflow engine enables easy workflow composition and execution.
- Gridway [37]. A cross infrastructure meta scheduling framework, hiding complexity of the underlying Grids.

Grid Portal is yet a higher level abstraction, which requires a little, if any, installation and configuration. Now users could access Grid by simply using a web browser. Some Grid portal examples are:

- Teragrid portal [38]. Teragrid users' portal. Information service regarding the Teragrid is provided. It also has file operation functionality over the resources that the user have access to. A GSI-SSH terminal is used to login the resources through inside the web page.
- Gridsphere [39], OGCE portal [31]. Two similar projects, both JSR168 compliant frameworks, provide most of functionalities to access Grid.

From the evolving of the approaches we could summaries the road to access Grid as a process of abstraction in which for each transition a newer toolkit reuses the exist toolkit and builds upon them while providing more friendly user interface and easy-to-use application interface.

Globus Toolkit provides infrastructure and command line tools to access Grid. Java CoG Kit, while provides command line tools for the similar functionalities, extends the functionalities to support more feature such as GUI based workflow composition. Gridsphere and OGCE, on another hand, introduces JSR-168 based portal framework and portlets, from which users can access Grid from a web portal. Teragrid also provides a web portal for Grid users, from where users could manage their accounts, monitor resources, and access to Grid through a GSI-SSH terminal.

While providing the same or similar functionalities, if not more, a newer generation of toolkit or framework requires less time to install and configure. Web portal based technologies free users from the burden by requiring only a little to setup from a client.

The proposed Web 2.0 technologies based JavaScript abstraction framework and portal use newer technologies and follow newer design philosophy and patterns like SOA and AJAX based interaction between web client and server. It hides the complexity to interact with Grid and cyberinfrastructure on the backend service tiers, and requires zero installation in client side. This essentially becomes a Software as a Service(SaaS) solution.

3 Design

In order to design the framework and a JavaScript library for the Grid the following requirements are identified as essential features:

- *Low installation footprint* is necessary to support fast downloads as well as an easy setup and maintenance through a small manageable code base.
- *Ease of use* is important to make the JavaScript based API and interfaces useful for Grid and Web developers.

- *Security* is needed to gain access to Grid resources in order to avoid compromising the system. This is especially important for web applications.
- *Basic Grid functionality* must be provided in order for developers to create Grid-based client applications.
- *Advanced functionality* is needed as many developers do not want to replicate functionality provided by other Grid middleware and upperware. This includes more sophisticated job management functionality, workflow queues, and the availability of elementary graphical user interfaces (GUI).

To fulfill these requirements, the framework is methodologically designed as a layered multi-tier architecture. The architecture is comprised of several components. The most important components of the architecture are depicted in Figure 8 and more detailed info will be explained next. The architecture contains a Web client providing access to basic and advanced Grid functionalities, as well as the service stack to support the functionalities. To keep the footprint of the library small most of the functionalities in regards to the Grid are executed on a secure server. As this server mediates the tasks to the Grid, we refer to it as the *mediator service*.

First, the Web client provides a high-level application programming interface (API) to the Grid, a Grid workflow system, and components to access simple Grid functionality through graphical user interfaces. While using the API the developers can build portals specifically targeted for Grids and integrate customized JavaScript-based GUIs. Furthermore, the framework provides important functionalities such as authentication, file transfer, job and workflow management. These components will be deployed into an application server such as Apache Tomcat. They will be downloaded and run from within a web browser at runtime.

Then comes to the service stack behind the scene. All interactions to the backend Grid or cyberinfrastructure are conducted through the backend mediator service. The mediator service is responsible for communicating with Grid services through underlying Grid middleware such

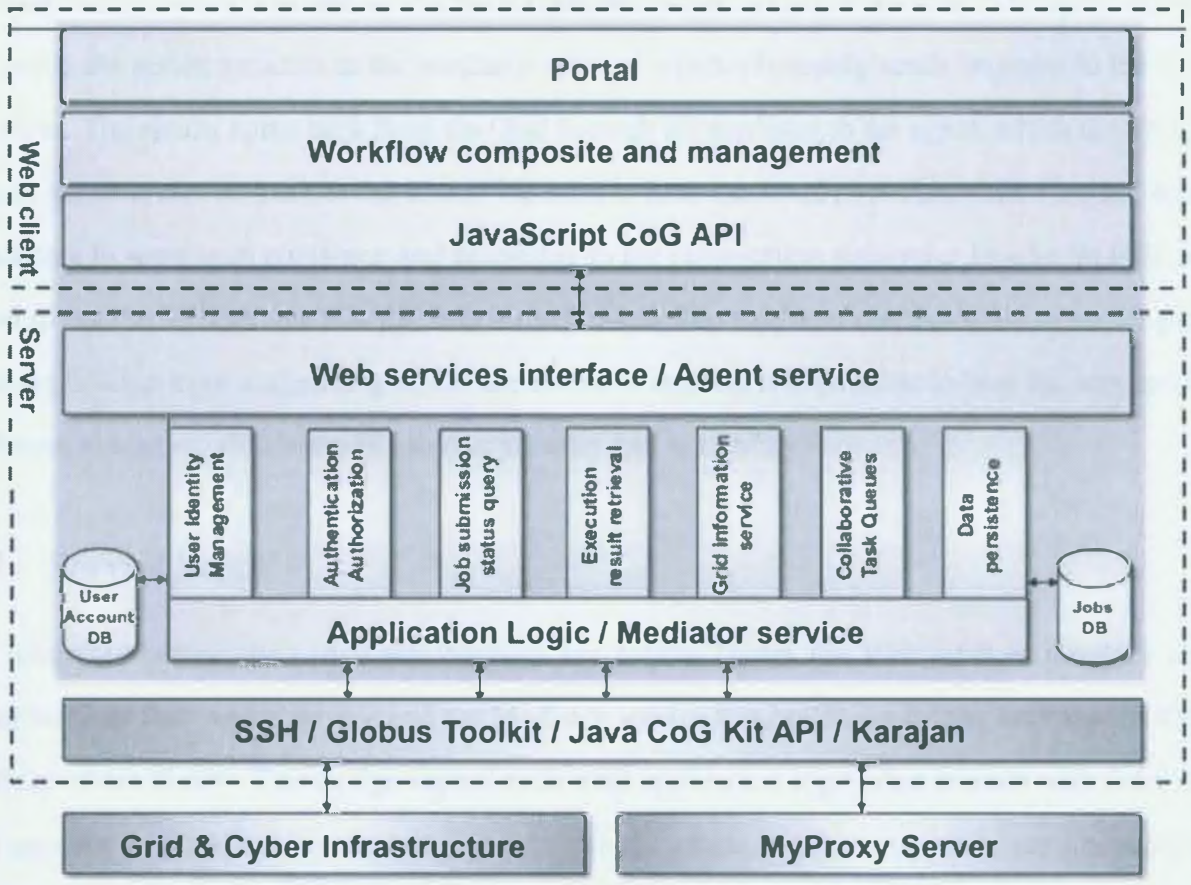


Figure 8: System Architecture

as the Java CoG Kit, Globus toolkit, or even SSH. Meanwhile it exposes the Grid services via standard Web services. It allows the use of a personal queue management mechanism that keeps track of all jobs and workflows submitted to the Grid in a more convenient form than the current generation of Grid middleware, such as the Globus Toolkit. The mediator service contains several essential functional modules such as user account management, authentication, job and workflow management, collaboration queues, and a persistence module allowing users to keep track of their jobs and tasks on the Grid.

An intermediate component exists called *agent* service that handles all communications between the web client and the mediator service. This means the agent service acts both as a service

provider to the web client and as a service consumer of the mediator service. The agent service forwards the action requests to the mediator service, which ultimately sends requests to the Grid services. The results come back from the Grid through the mediator to the agent, which in turn forwards the information back to the client. We have to host the JavaScript files, CSS files and other resources in some web container, and according to the same-origin policy for JavaScript [40], we need to put the web service that the JavaScript calls in the same web container. By separating the real application logic and putting it into the mediator service, it is possible to host the services on different machines, which would increase security and scalability.

3.1 Service Stack

As described before, the server side contains two logical layers, the Web services interface also referred to as user Agent service and the Mediator service that builds the bridge between the Grid and our client library. This design separates the real application logics that interact with Grid and the services which should be hosted together with the client in some web container like Apache Tomcat. This increases the system security and scalability.

3.1.1 Mediator service

The *mediator service* is where the application logic code resides. It contains several modules to deal with different functionalities and offers a persistent view of interactions with regular Web services, Grid services, or authentication. While the agent service just forwards all requests from the user to the mediator service, the mediator service supports multi-user concurrency by maintaining session and state for each user. A persistent database is used to maintain state about workflows and collaboratively managed queues. It is important to recognize that our notion of compute tasks exceeds that of the basic Globus functionality. It also allows one to integrate services into the tasks managed by a user that are not typically executed by Globus. Thus a query to Web services offered by another organization like Google can be readily abstracted a task with its own status. Hence,

our task model includes not just tasks that are executed on Globus enabled Web services.

3.1.2 Agent service

The *agent service* functions as an intermediate service between the Web client and the mediator service. It hides much of the complexity of the Grid and allows for the deployment and integration of resources where the installation of Grid middleware is not possible. Hence, it works as a proxy for users to interact with the mediator service, and through it to Grid services. It can perform proxy credential delegation for users to authenticate to the Grid and to provide single sign on. The agent uses web services to provide functionality calls and communication with web client. Thus, the agent service itself is a service provider for the client but also a service consumer of the mediator service.

3.2 Collaborative Queue

Besides the typical Grid functionalities such as authentication, job management and file transfer, the design includes features that enable users to share and manage workflows among a group of users collaboratively. As such a *shared workflow* and a *shared queue* that can contain multiple workflows to be executed are defined. Through the concept of sharing user-based access control supported by ownership and group based access control supported by membership in a participant list are provided.

To fulfill this functionality, we have defined the objects representing a shared queue and a shared workflow:

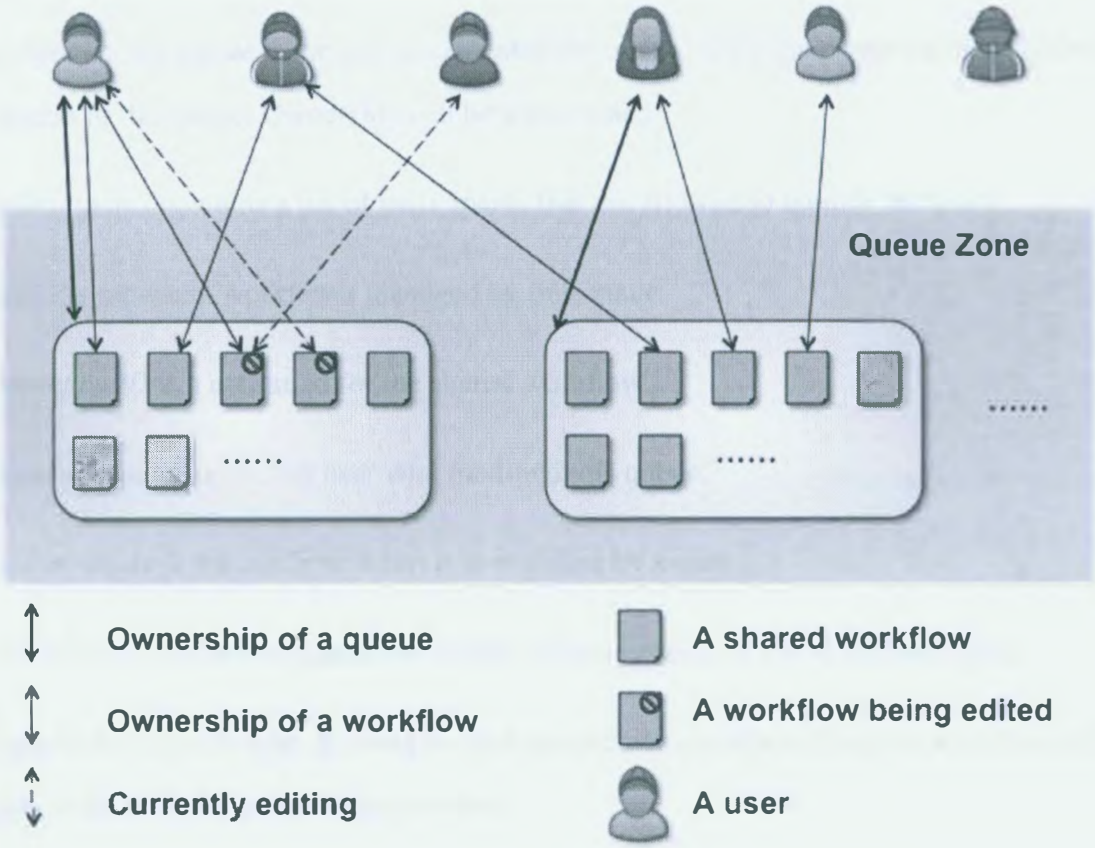


Figure 9: A scenario for a shared Queue.

$$queue ::= (queueID, label, owner, participants, objects)$$

$$workflow ::= (workflowID, label, owner, lastmodbyuser, lastmoddate, writeToken, type, WFObj),$$

The attributes to these objects are as follows:

- *label* is a field to store a meaningful label to identify the object easily.

- *QueueID* is a unique ID for the queue.
- *owner* for the *queue* is the one who created the queue. Only the owner can grant other users access to the queue. Ownership can be transitioned.
- *participants* contains a list of participants that are allowed to modify the queue.
- *objects* represent workflows managed by this queue.
- *workflowID* is a unique id for the shared workflow.
- *lastmodbyuser* is the last user who modified this queue.
- *lastmoddate* is the last time when it is modified by a user.
- *writeToken* is a lock to guarantee atomic write operation of the workflow object.
- *type* is the object's type. It could be system executable script or Karajan workflow, a Globus job, a file transfer, to name only a view.
- *WFObj* is the workflow description wrapped in this workflow object.

It is obvious from the definition of the attributes to that it will be possible to provide a shared workflow. As pointed out before, it is important for the activities typically conducted within an ad-hoc virtual organization to coordinate computational experiments as part of group activities. To support the notation of Workflows we have introduced a number of supporting APIs that makes the development of workflow related tasks easy for clients and services.

3.3 Web Client

The Web client is where Grid developers develop Grid web applications or science gateways based on the Cyberaide JavaScript API. The API provides the elementary functionalities to access the

Grid, while users could utilize other JavaScript libraries or integrate their own to build Web 2.0 Rich Internet Application (RIA) for Grid.

As an example and proof of the toolkit's usability, a simple portal user interface is included. The portal fulfills the tasks by using the JavaScript API allowing access to a number of essential Grid services. These services provide the following functionalities to the user:

1. Creation of jobs and workflows on the client side;
2. Job and workflow composition and submission for execution.
3. Information queries and monitoring of the status of jobs and workflows.

It is important to note that at this time Globus GRAM does not support managing jobs by multiple users. We are able to provide this functionality because all user jobs are managed through the mediator service that allows jobs to be executed through a service in behalf of a coordinating user.

One of the most important feature for many users will be the definition of APIs in JavaScript dealing with job files, and security management. Based on the lessons learned from the Java CoG Kit, we designed APIs in JavaScript that address the well-established functionalities needed by many users [41]. This includes tasks that manifest themselves as authentication, file transfers, jobs, status queries, workflows [42]. The essential of the API is illustrated and explained as in Table 1.

Please note that although JavaScript is not a strong typed language and does not provide a general mechanism for defining a class or object-type definition [25], it is a well-established practice to define custom objects in JavaScript that behave, in many ways like classes in Java. It achieves these through the *prototype* mechanism, and provides most of what Object Oriented language could achieve. Hence we use the term JavaScript classes throughout the table and rest of the text.

We have divided the application programming interface(API) into multiple categories. These

Table 1: Application Programming Interface

Class	**org.cyberaide.js.jsExecutable – functions related to Job and Workflow Abstraction**	
		Object with a number of attributes that abstracts the concept of execution of jobs.
Class	**org.cyberaide.js.jsAuthenticator – functions related to Security**	
o	authenticate (callback)	Authenticate through the attributes by using the provider defined in the jsAuthenticator object.
Class	**org.cyberaide.js.jsUtil – functions related to User Job Management**	
o	authenticate(jsAuthenticator, callback)	The jsAuthenticator's authenticate() function will be called. For example, if we use in the attribute provider as part of the jsAuthenticator class attributes the value MYPROXY, then a myproxy authentication will be carried out.
o	submit(jsExecutable, callback)	Submit an Executable to server for asynchronous execution.
o	transfer(source, destination, callback)	transfer data from source to the destination each of which are defined as URI. A transfer itself is packaged as a submit function and is treated the same way as other executables are.
o	list(callback)	List all the jsExecutables submitted to the server.
o	queryStatus(executablesids, callback)	Get state of the set of executables specified by their ids.
o	queryOutput(executableids, callback)	Get list of all the output files the set of executables specified by their ids.
o	getOutput(executableids, resultFile, callback)	Get back the content of one output file of the specified executable ids.
Class	**org.cyberaide.js.jsWorkflow – functions related to Client Side Workflow Management**	
o	addJob (name, jsExecutable)	A new job with a given name is added to this workflow.
o	deleteJob (name)	The job with the name specified by parameter job name is deleted. As a result, all related dependency is deleted as well.
o	listJobs( )	List all jobs in a workflow.
o	addDependency (parent, child)	Add dependency between a job patent and the child.
o	removeDependency(parent, child)	Remove dependency between a job parent and child.
Class	**org.cyberaide.js.jsQueue – Server Side Shared Queue Management**	
o	listQueues(callback)	List all the queues that the user are participating.
o	grantAccess(queueID, userlist, callback)	The owner of a queue can give some other users access privilege to the queue.
o	add(queuename, jsWorkflow, callback)	adds a workflow to the queue to be executed.
o	remove(queueID, workflowID, callback)	The owner of a workflow can remove it.
o	list(queueID, callback)	list all workflows' metadata shared in the queue.
o	listParticipants(queueID, callback)	List all the participants of the queue.
o	listByName(queueID, username, callback)	List the workflows' metadata owned by a user from one queue.
o	listByType(queueID, provider, callback)	List all the workflows' metadata with the specified type, say, Karajan workflow, from the specified queue.
o	getStatus(queueID, workflowID, callback)	Query the specified workflow's status. It could be locked since being edited by a user.
o	browseWorkflow(queueID, workflowID, callback)	Download the workflow to client side to browse.
o	obtainWriteToken(queueID, workflowID, callback)	The user try to obtain the write token.
o	editWorkflow(queueID, workflowID, callback)	The user will try to obtain the write token and then to download the workflow to edit.
o	updateWorkflow(queueID, workflowID, WFObj, callback)	Update the modified workflow when complete editing.

include JavaScript objects dealing with jobs, workflows, and authentication. In addition we provide functions that are used as part of workflow management, job management, and queue management.

The org.cyberaide.js.jsExecutable class is used to define an object that will be executed in the Grid, which could be a single job or a number of jobs that are part of a larger workflow. In summary, the class has two important fields: a *provider* and *attributes*. A provider could have the values *system*, *cog-karajan* or any other value referring to a workflow engine assuming that support for the workflow type exists within the gridshell. Attributes describe properties of a task to be executed.

The org.cyberaide.js.jsAuthenticator class is used to wrap an object that handle the authentication in the JavaScript. It has a *provider* and *attributes* fields and uses an authenticate() function to initiate the authentication with the server. The provider represents the authentication method to be used. Currently the *MyProxy* authentication is supportted. The attributes contain the necessary information to authenticate the user while using the method provided in the Provider field. For MyProxy authentication, the attributes should contain the MyProxy server's hostname, port, username and corresponding passphrase to retrieve the proxy credential.

The org.cyberaide.js.jsUtil class uses the other defined classes to perform the interactions with the Grid services, through the agent service and Mediator service. The callback functions has a single parameter *jsonRet*, which contains a JSON [43, 44] formatted object containing the response. To avoid security issues this response is wrapped into a single object as discussed in [45–47].

The org.cyberaide.js.jsWorkflow class represents a *CoG Karajan* workflow. A *CoG Karajan* workflow can contain multiple jobs, and it supports hierarchical workflow, which means a workflow could be a *job* in another workflow. A user can manage jobs with dependencies between them as part of a single workflow.

The org.cyberaide.js.jsQueue class is used to define objects that deal with the interactions between the web client and the server side's collaborative queue management functionality. Users are able to store workflows composed on client side to the server side shared queue zone through

a jsQueue object. These workflows can also be managed and shared between users. Callback functions and the returned JSON objects share the same arguments as that in the jsUtil.

3.4 Security Considerations

Naturally, security is of utmost importance in the development of any portal framework to the Grid. It is important to identify possible security issues that may arise in a JavaScript-based solution.

Since HTTP is stateless, we need to maintain some method to record users' states. Typically this is done by using HTTP cookies [27] which may include user and/or session related information. However, due to the well-known Cross-site Scripting(XSS) [48, 49] and Cross-site Request Forge (CSRF/XSRF) [50] vulnerabilities, we avoid the use of cookies to minimize the potential risk of these attacks. Instead we use a security token that is assigned to an authenticated user in order to prove users' identity during the session.

Security tokens maintain session information in similar way to cookies. Since we maintain the token only inside a JavaScript object during a session, it is immune to cookie related attacks, which typically access the cookie using *document.cookie* [51, 52].

The security architecture is supported by the following features.

- HTTP over SSL/TLS (HTTPS) [53] is used between the web client and web server to gain transport layer security.
- WS-Security standard [28] is used to secure the Web service traffic between the web server and the underlying Mediator service.
- Grid Security Infrastructure (GSI) [54] is used between the mediator service and the Grid.

To better understand some of the security aspects we walk through the following scenario where a user authenticates using MyProxy and performs tasks such as job submission, status query and result retrieval. In Figure 10 flows are corresponding to (A) user authentication request, (B)

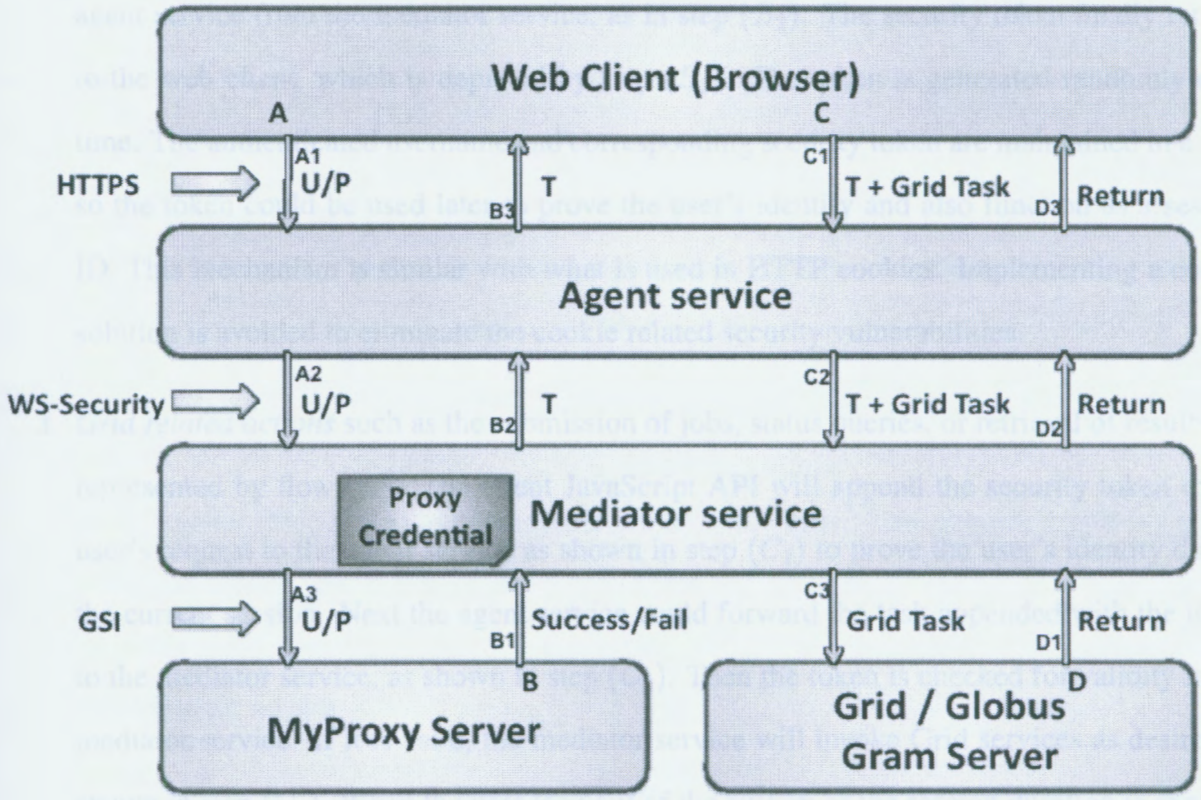


Figure 10: Security view of a typical use scenario

authentication response, (C) Grid related actions request and (D) Grid responses. A flow consists of multiple steps that are indicated by the inclusion of a number for the step.

1. An *authentication request* is shown by flow (A). A user tries to login to the system by providing the username/password (U/P) in the web client as in step (A_1). HTTPS guarantees that the traffic is secured between the browser and the agent service. The agent service then forwards the user credential to the user authentication module in the application logic layer (A_2). The authentication module could be used together with a user account management module or an external federated identity system such as MyProxy as in step (A_3).
2. An *authentication Response* of the authentication request from server is conducted as part of

Flow (B). If the user provided a valid credential, then a security token (T) is returned to the agent service from the mediator service, as in step (B_2). The security token finally returns to the web client, which is depicted by step (B_3). The token is generated randomly each time. The authenticated username and corresponding security token are maintained in a map so the token could be used later to prove the user's identity and also function as a session ID. This mechanism is similar with what is used in HTTP cookies. Implementing a cookie solution is avoided to eliminate the cookie related security vulnerabilities.

3. *Grid related actions* such as the submission of jobs, status queries, or retrieval of results are represented by flow (C). The client JavaScript API will append the security token to the user's request to the agent service as shown in step (C_1) to prove the user's identity during the current session. Next the agent service could forward the task appended with the token to the mediator service, as shown in step (C_2). Then the token is checked for validity in the mediator service. If it is valid, the mediator service will invoke Grid services as desired as shown in step (C_3). When the user logs out of the system or the session duration is expired, the security token will be invalided and recycled by the Mediator service and no further actions will be carried on to the underlying Grid services.

4. *Grid responses* that occure during task submission are shown in flow (D). The responses of the Grid related action requests are forwarded all the way back up to the client, which is shown in steps $(D_1)through(D_3)$.

4 Implementation

The system is based on object-oriented model and Service-Oriented Architecture (SOA) [4]. Entities in the system are all objects, while the SOA is used for the interaction between the distributed objects. The benefit of this approach is that it enables a loose coupling between distributed objects.

Thus, our concern is how to manage contracts between services and not the actual implementation details of the objects that may be implemented in different languages and frameworks. As long as we keep the contract between entities, changes on implementation of one part would not affect other parts of the system.

Java is used to develop server side application logics and Web services. After comparing the Apache CXF framework is chosen to facilitate the service stack development. CXF supports JAX-WS quite well, so we could use standard Java SE 6 web service technologies to write code, while using the tools from the CXF framework to assist the development and the deployment of the Web services.

Apache Maven [55] is used to manage the project code base. This is quite useful for such a project that has complicated Jar dependencies and building process.

4.1 Mediator service

In the implementation of the mediator that interacts with the Grid, we use the Java CoG Kit API and command line tools. Thus, through the Java CoG Kit we can enable proxy credential retrieval and delegation. In more detail, the Java CoG Kit JGlobus module provides functionalities to handle the interaction with MyProxy [56]. As long as a user stored his/her credential on a MyProxy server, the user can retrieve the credential by providing the username and user phrase at time when the proxy credential is generated. Thus the user delegates the application server to communicate with the underlying Grid infrastructure during the user session without the need to authenticate each time a job is submitted or a status query is issued.

To expose this service to the client, it is implemented and deployed as web service. WS-Security is used to secure the service traffic, either by Username-Token mechanism or PKI mechanism. Apache CXF framework [57] is used to develop and secure the mediator service due to its simplicity. For the Username-Token approach, we need some method to maintain valid user accounts. The user account info could be maintained in several states, either through a file or a

persistent database. A Java interface is defined to achieve this. Implementing this interface with different mechanism could achieve different management schema.

```
/*
 * IUserAccount.java
 */
package org.cyberaide.account;

/**
 * Define user account management interface for WS-Security's UsernameToken
 * solution
 */
public interface IUserAccount{

  /**
   * add a user
   *
   * @return true for successful operation and false otherwise
   */
  public boolean addUser(String username, String password);

  /**
   * delete a user
   *
   * @return true for succseeful operation and false otherwise
   */
  public boolean delUser(String username);

  /**
   * change password for a user
   *
   * @return true for successful operation and false otherwise
```

```
    */
    public boolean changePasswd(String username, String newPasswd);

    /**
     * reset user's password
     *
     * @return a string represents the new password
     */
    public String resetPasswd(String username);

    /**
     * check user credential's validility
     *
     * @return true for valid credential and false otherwise
     */
    public boolean isValid(String username, String passwd);

    /**
     * get a user's current password
     */
    public String getPassword(String username);
}
```

Tools and APIs from the Java CoG Kit and the Globus toolkit are used to establish the interaction with Grid services. Furthermore, the Java CoG Kit Karajan workflow framework is used to support workflow composition.

WSDL document is the standard way to define and describe a Web service. A Web service version of the *Hello World* example could be described as this:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="HelloWorld"
```

```
targetNamespace="http://www.example.org/HelloWorld/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:tns="http://www.example.org/HelloWorld/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
<wsdl:types/>
<wsdl:message name="sayHelloRequest">
    <wsdl:part name="sayHelloRequest" type="xsd:string"></wsdl:part>
</wsdl:message>
<wsdl:message name="sayHelloResponse">
    <wsdl:part name="sayHelloResponse" type="xsd:string"></wsdl:part>
</wsdl:message>
<wsdl:portType name="IHelloWorld">
    <wsdl:operation name="sayHello">
        <wsdl:input message="tns:sayHelloRequest"></wsdl:input>
        <wsdl:output message="tns:sayHelloResponse"></wsdl:output>
    </wsdl:operation>
</wsdl:portType>
<wsdl:binding name="HelloWorld" type="tns:IHelloWorld">
    <soap:binding style="rpc"
        transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="sayHello">
        <soap:operation
        soapAction="http://www.example.org/HelloWorld/sayHello" />
        <wsdl:input>
            <soap:body use="literal"
            namespace="http://www.example.org/HelloWorld/" />
        </wsdl:input>
        <wsdl:output>
            <soap:body use="literal"
            namespace="http://www.example.org/HelloWorld/" />
```

```
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="HelloWorldService">
        <wsdl:port name="HelloWorldPort" binding="tns:HelloWorld">
            <soap:address location="http://localhost:8080/HelloWorld" />
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

However it is not a practical way to define a Web service starting from writing the WSDL document. Instead, we could define a Java interface file with appropriate annotations.

```
/*
 * IMediator.java
 * @version: $Id v1.0$
 */
package org.cyberaide.ws.mediator;

import javax.jws.WebService;
import javax.jws.WebMethod;
import org.cyberaide.execution.*;

/**
 * The server side mediator to fulfill the main application logic
 * and expose them as web services
 */
@WebService
public interface IMediator
{
  /**
   * list files of the specified directory
```

```
    *
    * @param user the authenticated user
    * @param dir the relative dir, from the user's homeuser's home
    *
    * @return serialized filelist
    */
   @WebMethod(action = "listDir")
   public String listDir(String user, String dir);

   /**
    * retrieve a proxy credential from a myproxy server
    *
    * @param host myproxy server host
    * @param port myproxy server's port
    * @param user username
    * @param password passphrase to login to the myproxy server
    * @param lifetime the lifetime of the proxy credential (in seconds)
    *
    * @return true if retrieve successfully, false otherwise
    */
   @WebMethod(action = "retrieveCertificate")
   public boolean retrieveCertificate(String host, int port, String user,
       String password, int lifetime);

   ......//other methods omitted here due to page limit.
}
```

Popular JAX-WS frameworks such as the Apache CXF framework we used could generate WSDL document to describe the web service defined through a Java interface file. In this way we are still following the *Contract First* approach but defer the implementation of the application logics to later stage. A more important benefit is that, this way by changing the implementation

class file would not affect the *contract* we have agreed between the Web service provider and consumer.

4.2 Agent Service

We have implemented the agent service with the help of standard Web service technologies. Hence, SOAP [26] messages are used for communicating between the agent service and web client. The connection is secured through HTTP over SSL/TLS.

Presently, as we have done for the Mediator service, we use Java to develop and deploy the Web services, and we also use JAX-WS [58] annotations from Java SE 6 to specify the service.

The packaged web application archive is deployed in a web container such as Apache Tomcat [59].

To interact with the mediator service, the client stubs of the mediator service are generated by tools such as wsimport. The build process is controlled by Apache Maven as mentioned above to allow for easy deployment and upgradability. All Web page resources including static web pages, CSS files, the JavaScript CoG library, Portal JavaScript files, and necessary external JavaScript libraries and images are hosted within the same Tomcat server under the same HTTPS host name and port.

Secure traffic between agent service and web client is obtained through HTTPS connection, while WS-Security is used to secure the traffic with mediator service.

4.3 JavaScript Application Programming Interface

The API is implemented as a JavaScript file hosted together with the Agent service in a Apache Tomcat web container since the the JavaScript toolkit need to communicate with the Agent service. This is the requirement from JavaScript's same-origin policy, which defines that a JavaScript file from within a web browser could only talk to its origin defined by a host name and port number.

SOAP is the message format used by the JAX-WS web services. While we use SOAP as message format between web client and agent service, the output of the JavaScript APIs is transformed into JSON [43,44] format. Since in AJAX style programming the return values are handled as part of callback functions that can be customized by the developers, this allows us to easily integrate data into the JavaScript client program.

For example, when the client gets some response from the Agent service side that looks like:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
<ns2:listExecIdsResponse
    xmlns:ns2="http://agent.ws.cyberaide.org/"
    xmlns:ns3="http://mediator.ws.cyberaide.org/">
<return>
    2085137299_229231686_54913425_54913426_
</return>
</ns2:listExecIdsResponse>
</soap:Body>
</soap:Envelope>
```

The toolkit will parse the XML formatted message and retrieve the return value, and then code the value in JSON format like this:

```
{wfids:[2085137299, 229231686, 54913425, 54913426]}
```

So the API users could use the convenient JSON data directly in their callback functions to do whatever processing they want.

The goal is achieved through some mechanism within the toolkit that looks like this:

```
function listMyWfResponseInner(r, soapResponse){
    var ret;
    try{
        ret = soapResponse.getElementsByTagNameNS("*","return")[0]
                .childNodes[0].nodeValue;
    } catch (e) {
        listMyWfResponse(null);
        return;
    }
    var mySplitResults = ret.split("_");
    var numIds = mySplitResults.length - 1;
    var jsonRet = "{\"wfids\": [";
    for(i = 0; i < numIds - 1; i++){
        jsonRet += mySplitResults[i];
        jsonRet += ",";
    }
    jsonRet += mySplitResults[numIds - 1] + "]}";
    listMyWfResponse(jsonRet);
}
```

So upon received response from the Agent service the toolkit will process the message in an inner response callback function and then pass the JSON formatted data as the single parameter to the real callback function that API users predefined before calling the API. This is the typical AJAX style programming when develop web applications. It works in asynchronous mode and a user can pass a predefined callback function as parameter to a method call. The callback function will be automatically called in client side when a response from server side is available.

Some 3rd party open-source library is used with some modification to construct SOAP messages sent from client to the Agent service. For example, for the file transfer method, we may call this to construct the SOAP message with appropriate format and namespace:

```
var pl = new SOAPClientParameters();
pl.add("user", user);
pl.add("token", mySessionId);
pl.add("from", source);
pl.add("to", destination);
```

And use this to invoke the remote method exposed through Web service:

```
try{
    SOAPClient.invoke(agentURL, "transfer",
            pl, true, transferResponseInner);
} catch(e) {
    //
}
```

Behind the scene, the library will construct the SOAP message, do the serialization, and then invoke the remote method through XMLHttpRequest object [60].

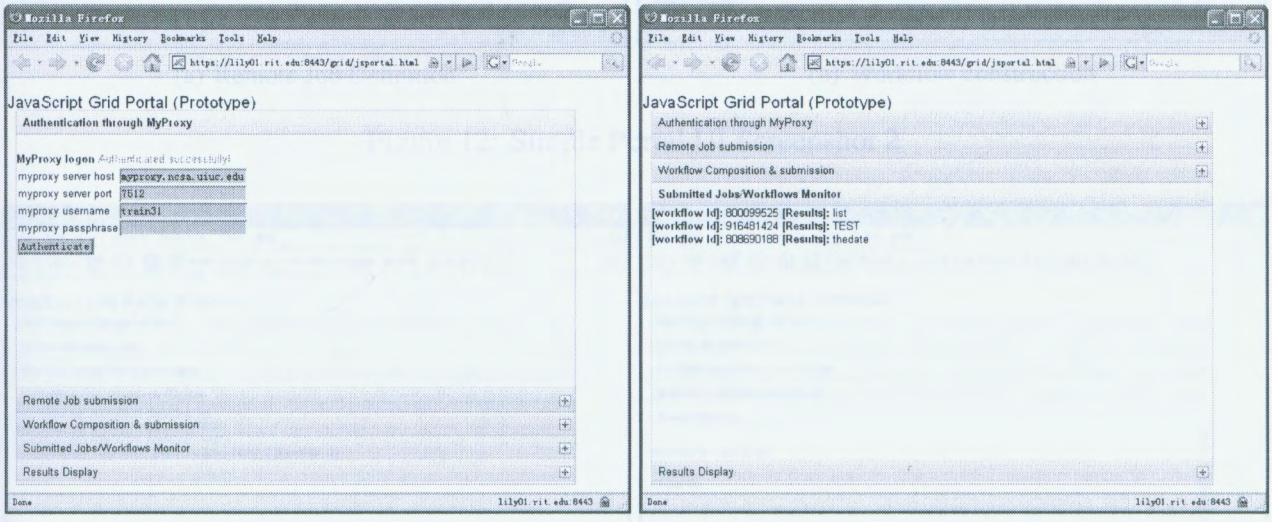
For example, a request constructed and sent by the toolkit from the client to the Agent service might look like:

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
<ns:listExecIds xmlns:ns="http://agent.ws.cyberaide.org/">
    <user>fuwang</user>
    <token>133221224</token>
</ns:listExecIds>
</soap:Body>
</soap:Envelope>
```

4.4 Portal As A Web Client

Built upon the Cyberaide JavaScript APIs, a simple portal UI (see Figures 11-13) is developed to provide generic purpose functionalities for access Grid service while communicating through the service stack.

External JavaScript libraries such as jQuery library [61], DOJO toolkit [62], EXT JS library [63], etc. can be used to ease the development of a user friendly graphical interface with desktop application comparable performance so we don't need to reinvent wheels.



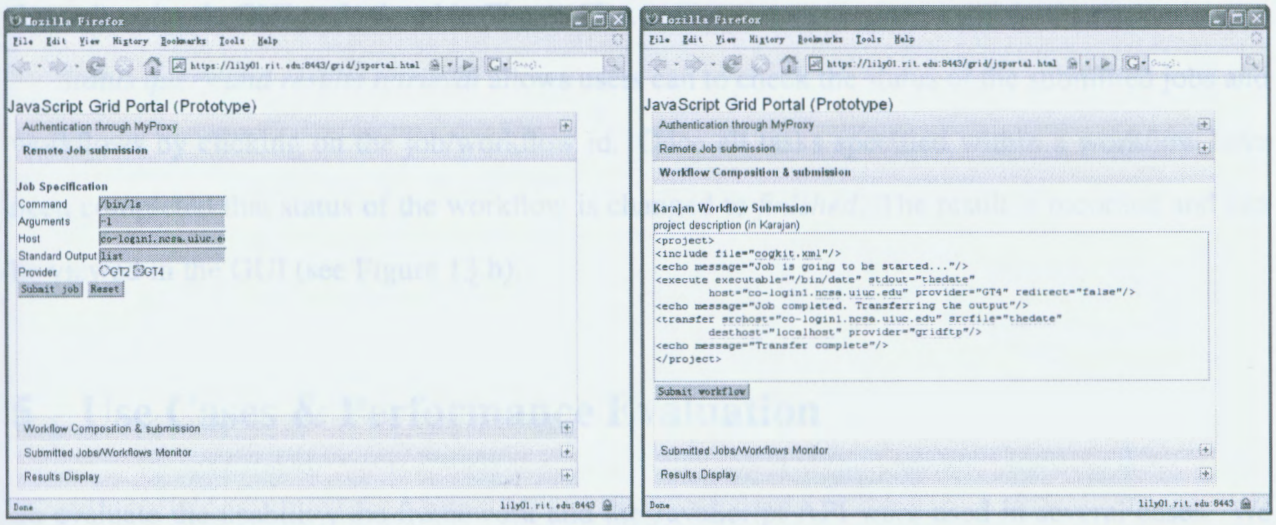
(a) User Login Page (b) History Jobs

Figure 11: Simple Portal UI Screenshot 1

User authentication is typically the first task. An external MyProxy authentication is used for user authentication. By providing appropriate MyProxy server settings and the right username and passphrase, users will be able to authenticate as shown in Figure 11.a.

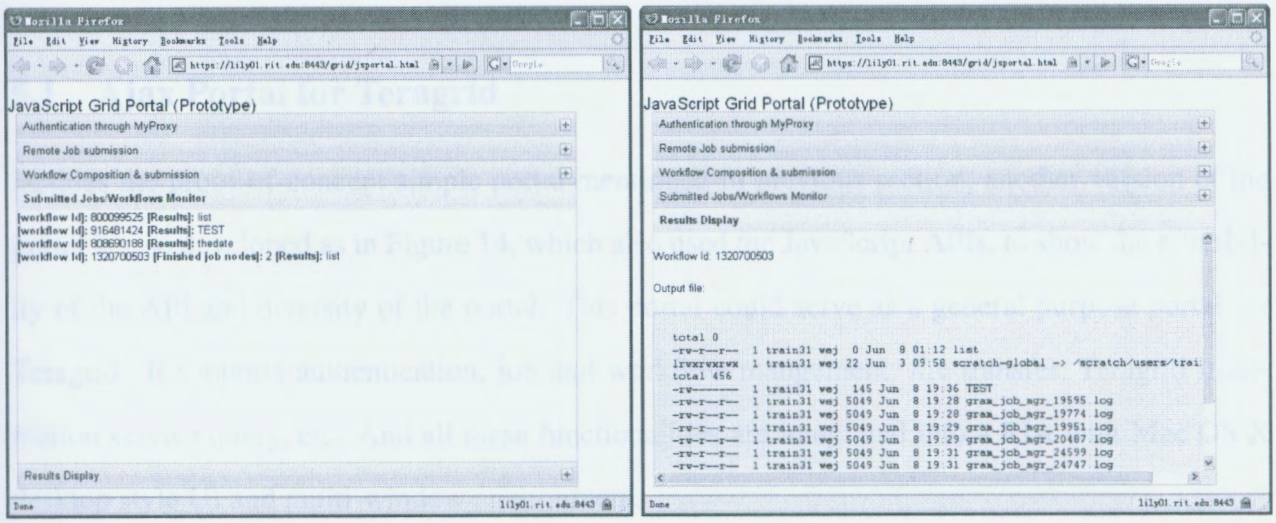
The *Histrory* of jobs allows us to check the history of submitted jobs/workflows as shown in Figure 11.b.

Job construction and workflow composition is supported by GUI for single job submission (see Figure 12.a) and for workflows (see Figure 12.b). For simple job specification users can fill out a form. For workflows we provide at this time a simple text window that accepts Java CoG Kit



(a) Remote Job Definition (b) Workflow construction

Figure 12: Simple Portal UI Screenshot 2



(a) Job Status Query (b) Result Retrieval

Figure 13: Simple Portal UI Screenshot 3

Karajan workflows.

Job execution is conducted once the job or workflow is specified. While explicitly pressing the submit button, the web client will invoke a number of web services through our mediator. Then a jobId will be returned to the client side as a handler. The execution of jobs can be monitored

through a simple GUI as depicted in Figure 13.a.

Status query and results retrieval allows users can to check the status of the submitted jobs and workflows by clicking on the job/workflow id. Once all tasks specified within a workflow have been completed that status of the workflow is changed to *finished*. The result is recorded and can be viewed in the GUI (see Figure 13.b).

5 Use Cases & Performance Evaluation

To evaluate the usability, the framework and the JavaScript API were used in several cases. We will look into some typical use cases in detail in this section. The performance is also evaluated and shown following the use cases.

5.1 Ajax Portal for Teragrid

Besides the proof-of-concept simple portal mentioned in previous section, another version of the portal is also developed as in Figure 14, which also used the JavaScript APIs, to show the reusability of the API and diversity of the portal. This portal could serve as a general purpose portal for Teragrid. It supports authentication, job and workflow mangement, file transfer, Teragrid information service query, etc. And all these functionalities are integrated into a RIA with Mac OS X desktop style UI and multi-window environment.

This portal work based on the entire Cyberaide JavaScript framework, and provides a RIA solution for the Teragrid portal. Because each Teragrid user has access to a login node, we can host the mediator on one of these nodes, as by default each Teragrid user will have access to such a node as it is part of the user account management of TeraGrid. The result is that the client has a zero install base and the web application will provide all the essential functionalities to access Grid.

More information about this portal are shown in Appendix as a portal user manual.

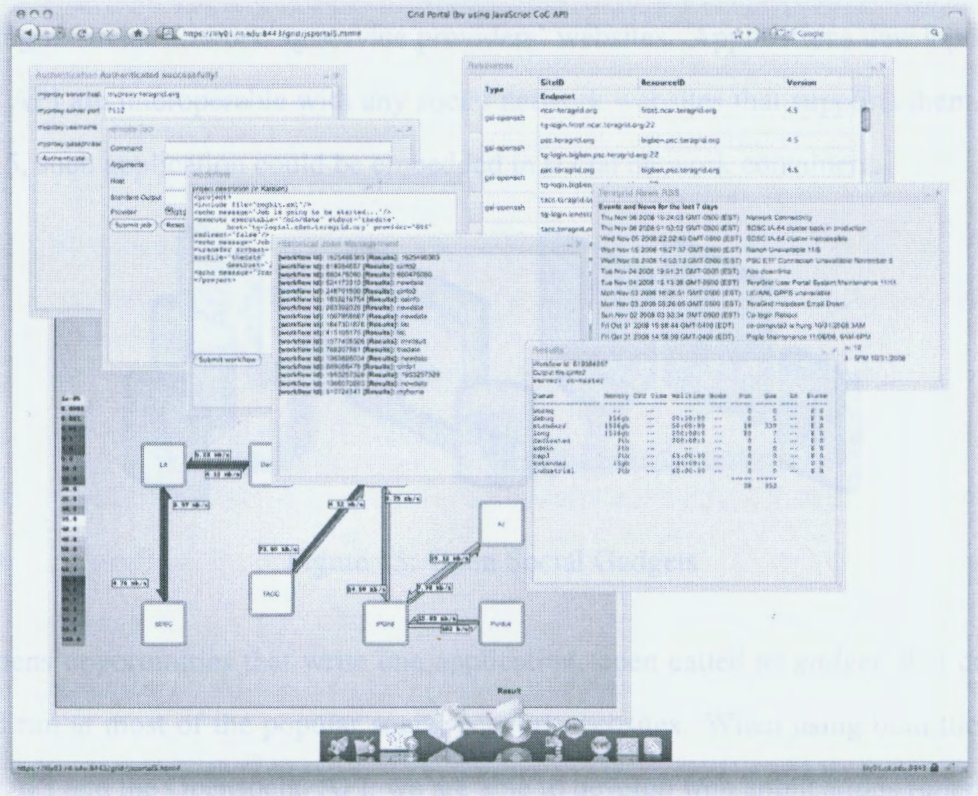


Figure 14: An alternative Portal

This use case shows the framework developed and the API provided could be used to develop RIA style generic purpose Grid portal, or domain-specific scientific gateways.

5.2 Opensocial Gadgets

In recent years we saw social network services booming. Social networking websites are changing the way we communicate and interact. Opensocial [64] provides a set of API for web applications running in social networking service providers' websites. Applications developed using the opensocial API are interoperable with any social network websites that supports them. As shown in Figure 15, such application could be embedded in social network containers.

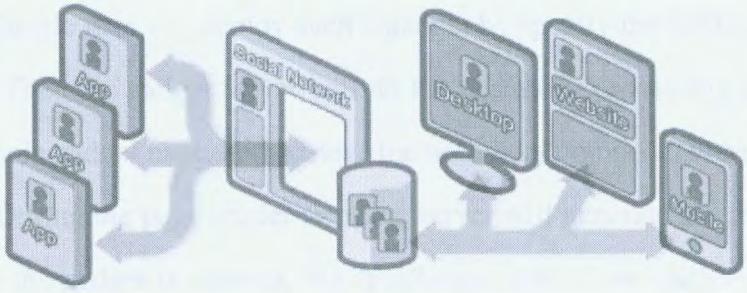


Figure 15: Open Social Gadgets

This opens opportunities that write one application, open called as *gadget*, that could be deployed and run at most of the popular social network websites. When using both the Cyberaide JavaScript API and the Opensocial API, we are able to develop web applications running at social network websites for cyberinfrastructures.

Figure 16 shows the architecture when the Cyberaide JavaScript framework and gadget developed using the Cyberaide JavaScript API are working in the iGoogle google gadget environment.

On the left side of the figure is essentially a Cyberaide JavaScript framework deployment. However when working with google gadgets, there are some important changes.

- The application(gadget) is defined using XML file and hosted in a web server.

- During runtime the gadget is communicating with its backend service stack indirectly through some proxy mechanism in the gadget server, or container.

When dealing with the gadget XML definition, we need only a little modification for a typical web client of the Cyberaide JavaScript framework to make it conform to the gadgets specification. To deal with the second issue, the original underlying communication mechanism between web client and the Agent service need to be modified accordingly. We need to replace the XMLHttpRequest based mechanism with the corresponding calls from the opensocial API, specifically the makeRequest() API call.

To deploy and use such a gadget with iGoogle gadget container is easy, as shown in Figure 16. First step, an iGoogle user can deploy such a gadget by specify the URL of the XML file that defines the gadget. Then the gadget server will do the necessary processing of the XML file and embed it into the whole gadget page, and retrieve the necessary components back from the original application server. During the page render step, the server will generate the rendered gadget page in an *iframe*. When the gadget is running, the JavaScript code of the gadget could be interacting with the original backend service and other third party resources through the proxy provided by the gadget framework.

An implementation of this architecture is shown in Figure 17 and Figure 18. The first figure showns two gadget, one for job management and the other for file transfer, are deployed and shown together with other gadgets in an iGoogle page. The second shows the file transfer gadget is running in *Canvas* mode, which means only the gadget itself is shown in larger window instead of crowed together with one user's all gadgets installed.

This use case of the framework shows the reusability of the framework and how easily could it be integrated with other web application technoligies to augment the functionalities provided by each one of them otherwise could not be achieved.

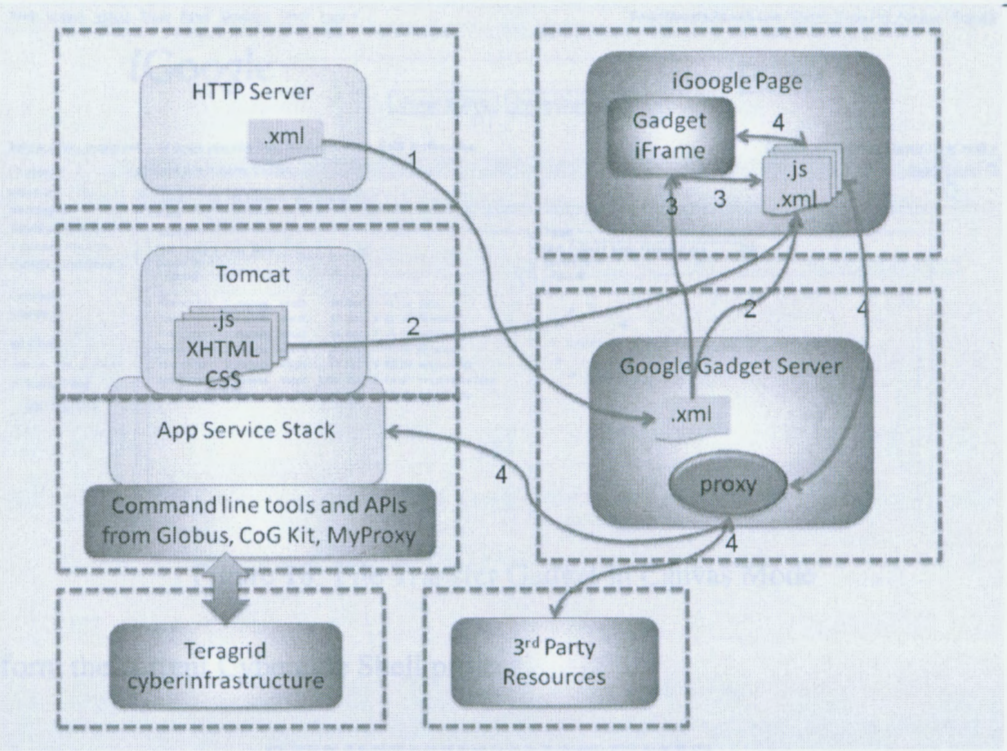


Figure 16: Cyberaide JavaScript When Working With Google Gadget

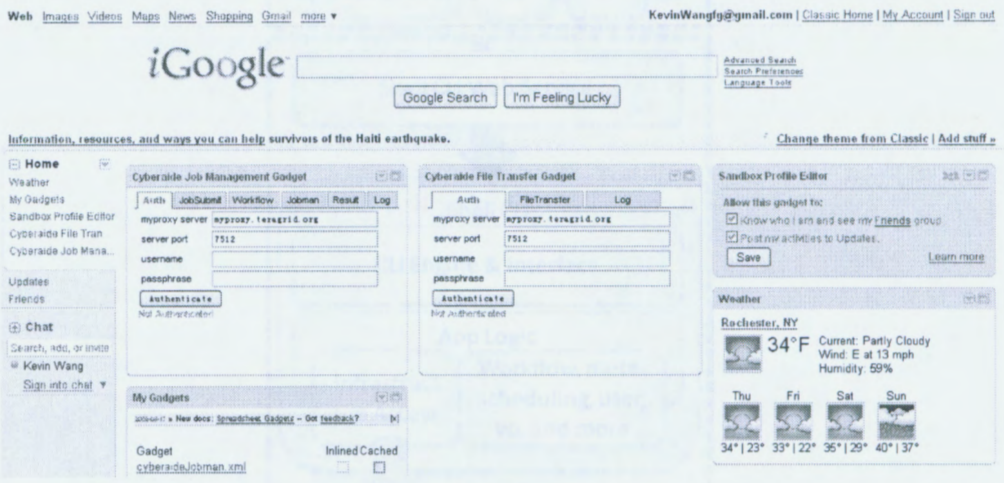


Figure 17: Gadget Version of Teragrid Job Management and File Transfer

5.3 Cyberaide Shell

When considering together with related projects within the bigger Cyberaide picture as shown in the Figure 1, the framework's architecture could be also served in other projects. One possibility

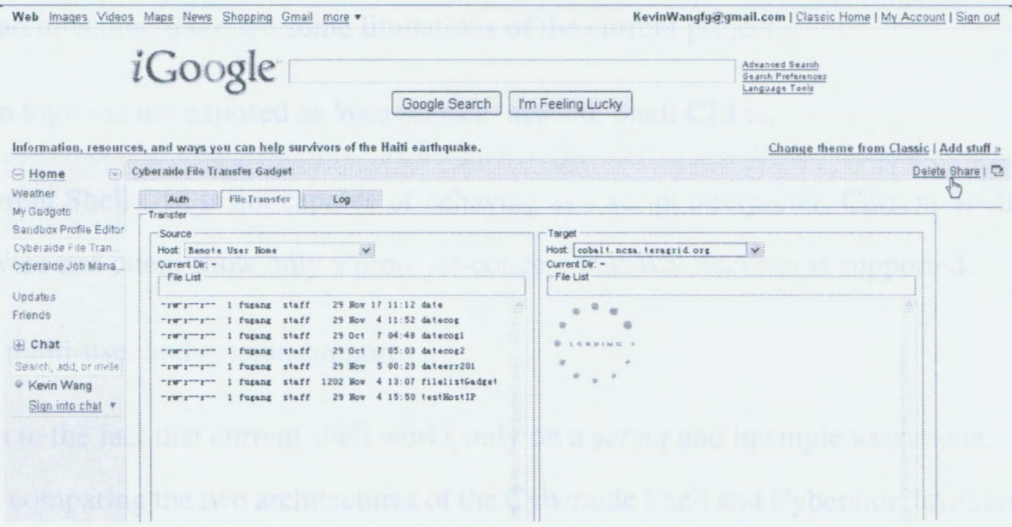


Figure 18: File Transfer Gadget in Canvas Mode

is to transform the current Cyberaide Shell project.

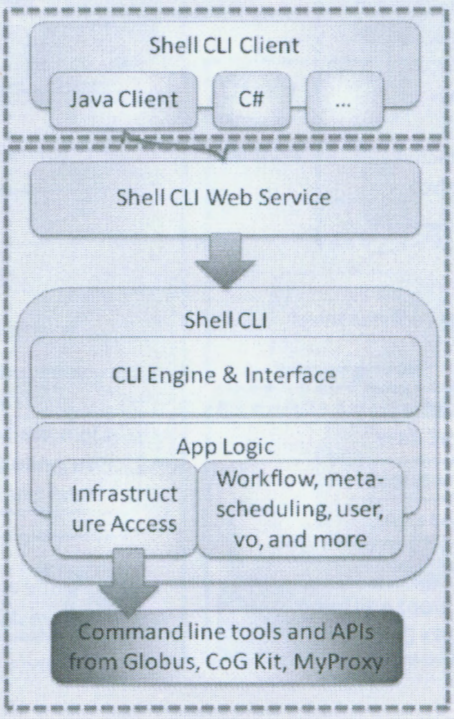


Figure 19: Cyberaide Shell Architecture

The current architecture of the Cyberaide Shell project is shown in Figure 19. As could be seen

from the architecture, there are some limitations of the current project.

- App logics is not exposed as Web service, instead, Shell CLI is.
- Current Shell CLI is not capable of behaving as a script interpreter. Current Shell CLI web service and client show only a proof-of-concept that WS-Security is supported.
- No multi-user and session support.

This leads to the fact that current shell works only on a server and in single user mode.

When comparing the two architectures of the Cyberaide Shell and Cyberaide JavaScript project as in Figure 20, we could see some possible approaches that could transform the current Shell project in a better shape.

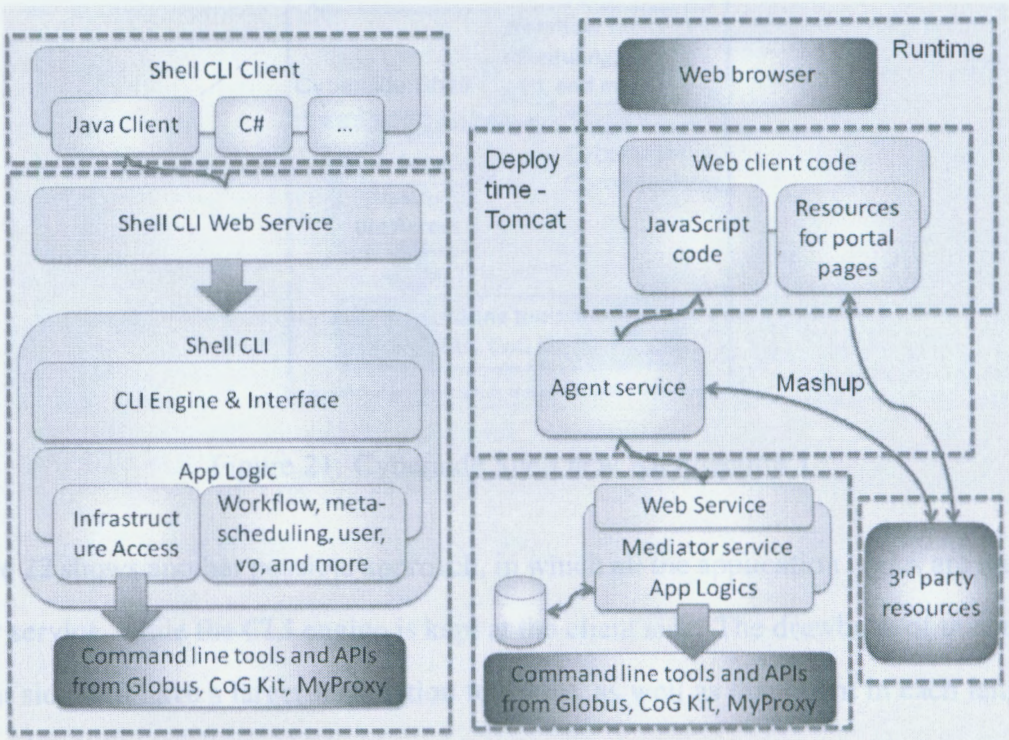


Figure 20: Architecutre Comparison of Cyberaide Shell and JavaScript

Figure 21 shows one possible transformation. During this approach, we split the original service/application logic tier to two tiers. The lowest tier shares the same functionalities with the

Mediator service from the Cyberaide JavaScript project. Then the Shell project could concentrate to the real application logics that provide add-on features like Virtual Organization(VO) and advanced scheduling. One drawback of this approach is that unless we could get a much stronger Command Line Interface(CLI) parser engine that could behaves as a script interpreter, we will not be able to get full set of feature from the client side.

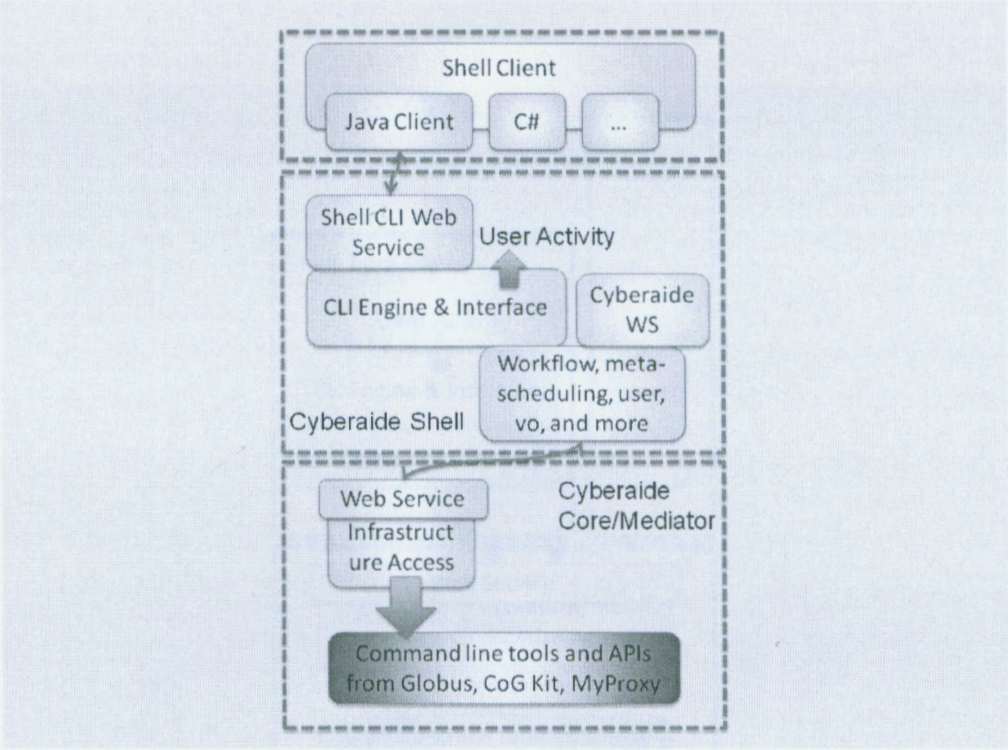


Figure 21: Cyberaide Shell new Architecutre 1

Figure 22 shows another possible approach, in which all the application logics are moved to the Mediator service, while the CLI engine is kept at the client side. The drawback of this approach is that client side will have a larger installation footprint, as well as that client in each language will have to have their own CLI engine. The benefit is that in this way we get a unified architecture that could serve both the Shell and the JavaScript project, as shown in Figure 23.

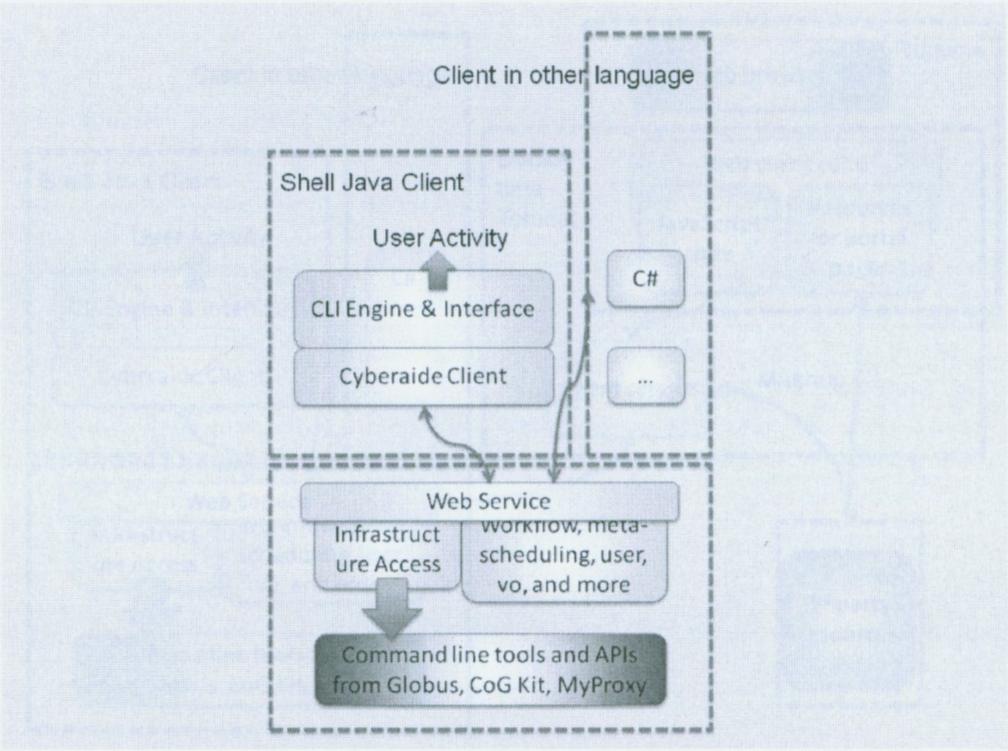


Figure 22: Cyberaide Shell new Architecutre 2

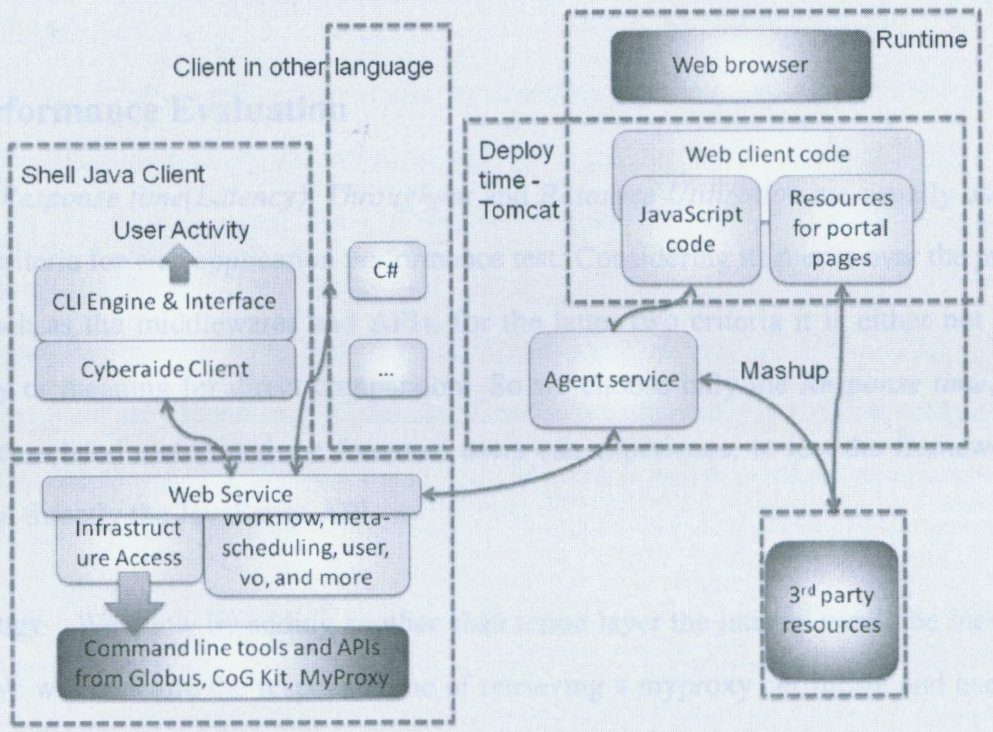


Figure 23: Cyberaide Shell Working under Unified Architeture with JavaScript

Other Possible Use Cases Besides the above mentioned use cases, the framework could also be beneficial to other cyberinfrastructure related projects due to its ability to extend the backend services and the easy integration with Web 2.0 techniques for RIA development. For example, when 'plug in' the Virtual Machine support in the Mediator service, the framework could be easily extended to enter the 'Cloud' world. As an update and augment of the current Teragrid, Futuregrid [65] could also utilize the framework to support the Web based applications development and deployment.

5.4 Performance Evaluation

Metrics *Response time(Latency)*, *Throughput* and *Resource Utilization* are usually used as performance criteria for web application performance test. Considering its merits over the precedence systems such as the middlewares and APIs, for the latter two criteria it is either not worse, or has no way or meaning for direct comparison. So we choose only the *Response time*, the most performance-related metric and the foremost users can experience, to test the framework, more specific and directly the JavaScript API.

Methodology We know by adding another abstraction layer the latency would be inevitably increased. We will measure the response time of retrieving a myproxy certificate and use the overhead introduced through the developed framework as a representative evaluation.

We will use n_0 to indicate the response time through the official myproxy Java API, in milliseconds; and use n_1 to indicate the response time through the JavaScript API and the developed framework while using a browser from the server directly; use n_2 to represent the response time through the JavaScript framework whiling accessing from a browser within a different network. Then n_1-n_0 will be the pure overhead introduced by the framework; and n_2-n_1 will be the overhead for typical usage scenario.

In Java code part, *System.currentTimeMillis()* will return the EPOC milliseconds. By instru-

menting this before and after the myproxy API call, it will measure the n_0 stated above.

In JavaScript, a similar method to return EPOC milliseconds is *(New Date()).getTime()*. By placing this just before the request and the first place when the callback function is called we could then measure n_1 and n_2.

Multiple times of experiments are carried on to find the statistical meaning of the testing results.

Testing Environment The Mediator service and Agent service are deployed into one physical server, iris01.rit.edu.

n_1 values are obtained through web browser accessing from the iris01.

n_2 values are obtained through web browser from machines in a different network.

The iris01 is a eight-core Mac Pro with 16G RAM installed, running OS X 10.5. It has Gigabit network connected to the RIT network.

Two machines with the Time-Warner cable RoadRunner service are used for the *different network* testing. Both are behind an NAT device. One named skwind is a netbook with 1.6G ATOM CPU and 1G RAM running Ubuntu 9.04, with 802.11g wireless network connection; another named skdell is a desktop machine with 3.0G Pentium4 CPU and 1G RAM running Microsoft Windows XP SP3, with 100M wired Ethernet connection.

The myproxy server in Teragrid *myproxy.teragrid.org* is contacted to handle the proxy certificate.

Results & Analysis Table 2 shows the overhead percentage results.

Table 2: Overhead As Percentage Of The Underlying API Execution Time(ms)

ExpNo.	*Mean(%)*	*Stdev(σ)*	*%80_Percentile*	*Mean(%)_%80_Percentile*	*Exp_Description*
1	3.1	0.2	3.2		iris01 client
2	33.9	27.9	26.2	22.1	skwind client
3	32.5	23.0	34.7	22.1	skdell client

The key messages here are:

- The pure overhead from the framework and API is negligible, showing only about a consistent 3%.

- For the normal use case, accessing from different network and machine, the overhead is larger but still remains a reasonable range. Specifically, 80% of the testing for each group has overhead less than 26.2% and 34.7% respectively, with an average 22.1% for those cases.

Table 3 shows the raw data obtained for the *same host* test.

Table 3: Performance Data For The Same Host Case

n	$Underlying\_API\_Time(ms)$	$JS\_API\_Time(ms)$	$JS\_Overhead(ms)$	$Overhead\_Percentage(\%)$
1	869	894	25	2.9
2	917	944	27	2.9
3	845	869	24	2.8
4	829	854	25	3.0
5	850	875	25	2.9
6	796	822	26	3.3
7	868	893	25	2.9
8	809	838	29	3.6
9	835	861	26	3.1
10	856	885	29	3.4
11	828	854	26	3.1
12	806	831	25	3.1
13	821	847	26	3.2
14	864	890	26	3.0
15	919	944	25	2.7
16	855	880	25	2.9
17	796	823	27	3.4
18	804	829	25	3.1
19	865	890	25	2.9
20	836	861	25	3.0
21	856	882	26	3.0
22	807	832	25	3.1
23	845	876	31	3.7
24	812	837	25	3.1
25	932	957	25	2.7
26	893	918	25	2.8
27	849	875	26	3.1
28	806	834	28	3.5
29	885	909	24	2.7
30	878	904	26	3.0

This shows how we get the raw data and get the overhead percentage as shown in Table 2.

To get a better view of the results, please see Figure 24, 25, 26.

Figure 24 shows the data distribution of the pure overhead of the JavaScript framework, with *average* value, *average*$\pm\sigma$, and the 80% percentile lines shown. Only a negligible 2%$\sim$4% overhead introduced if considering only the framework but not the more steps introduced through web

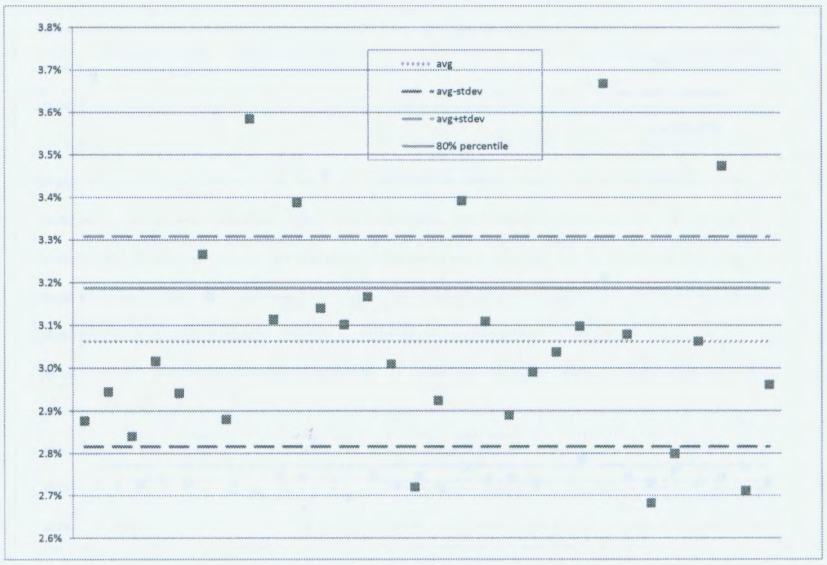


Figure 24: Overhead of the JavaScript API - Same Host

access for typical use scenario.

Figure 25 and Figure 26 shows the data distribution of the normal use case overhead, both with the all data average, 80% percentile line and the average of those under 80% percentile, which shows that most likely(under 80% probability) a user expect about 20% overhead for the response time comparing the direct Java API call. This translates to about 200ms, a very acceptable number when trading off with merits obtained through the framework. While these numbers could be vary and they are dependent mostly on users' network connection condition, the results obtained through the testing provide for us a general picture how the framework performs.

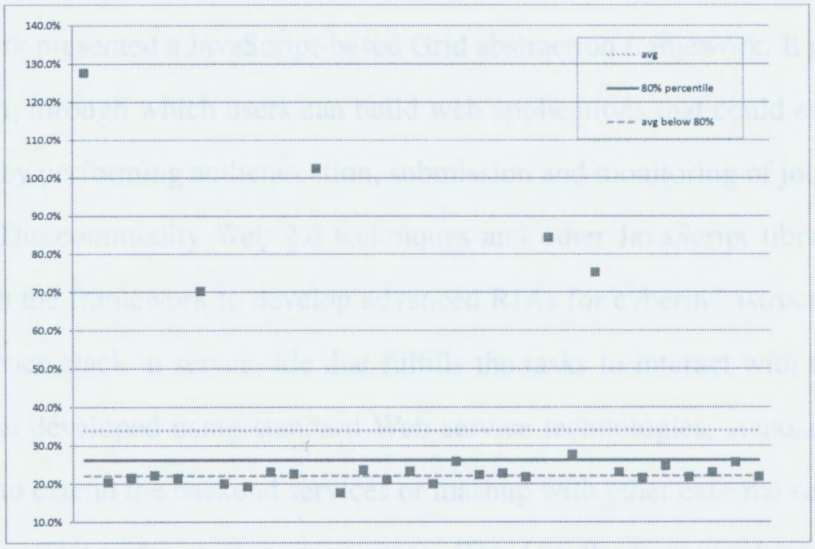


Figure 25: Overhead of the JavaScript API - NAT Wifi

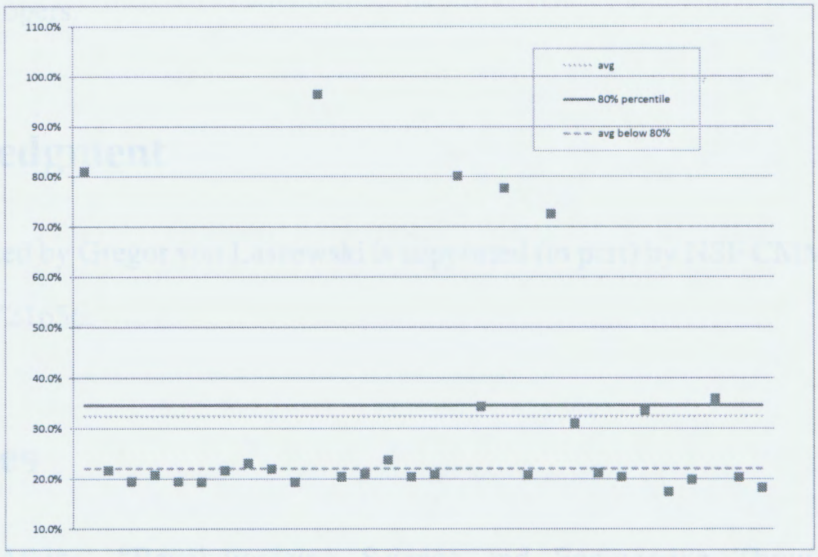


Figure 26: Overhead of the JavaScript API - NAT Wired Network

6 Conclusion

This thesis work presented a JavaScript-based Grid abstraction framework. It provides a client side JavaScript API, through which users can build web applications that could easily access the Grid infrastructure by performing authentication, submission and monitoring of jobs and workflow, and file transfer. The commodity Web 2.0 techniques and other JavaScript libraries could be easily integrated with the framework to develop advanced RIAs for cyberinfrastructure. The framework includes a service stack in server side that fulfills the tasks to interact with the Grid fabric. The service stack is developed using standard Web service technologies, standards and frameworks, thus it is easy to extend the backend services or mashup with other external services and resources while has little affect to the system architecture. Based on the framework and the JavaScript API provided, a user friendly Teragrid portal is developed from which users could access Teragrid from within a browser without the need of tedious installation and configuration works otherwise existed. Google gadget version Grid applications are also introduced as a development example for Grid developers.

Acknowledgment

Work conducted by Gregor von Laszewski is supported (in part) by NSF CMMI 0540076 and NSF SDCI NMI 0721656.

References

[1] D. E. Atkins, "Revolutionizing Science and Engineering Through Cyberinfrastructure: Report of the National Science Foundation Blue Ribbon Advisory Panel on Cyberinfrastructure," NSF, Arlington, VA, NSF Report, 2003. [Online]. Available: http://www.nsf.gov/od/oci/reports/atkins.pdf

[2] "TeraGrid," 2001. [Online]. Available: http://www.teragrid.org/

[3] "The Commodity Grid (CoG) Project." [Online]. Available: http://www.cogkit.org

[4] "OASIS SOA Reference Model." [Online]. Available: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm

[5] "The Globus Toolkit." [Online]. Available: http://www.globus.org

[6] "Cyberaide," Web Page. [Online]. Available: http://cyberaide.org

[7] I. Foster and C. Kesselman, *The Grid: blueprint for a new computing infrastructure*. Morgan Kaufmann, 1999.

[8] A. Uyar, W. Wu, H. Bulut, and G. Fox, "Service-oriented architecture for a scalable videoconferencing system," in *Pervasive Services, 2005. ICPS '05. Proceedings. International Conference on*, 11-14 July 2005, pp. 445–448.

[9] L. Zhang, J. Li, and M. Yu, "An integration research on service-oriented architecture (soa) for logistics information system," in *Service Operations and Logistics, and Informatics, 2006. SOLI '06. IEEE International Conference on*, June 2006, pp. 1059–1063.

[10] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, "Web Services Description Language," Tech. Rep. NOTE-wsdl-20010315, Mar. 2001, revision 1.1 - March 15, 2002. [Online]. Available: http://www.w3.org/TR/wsdl

[11] "Universal Description, Discovery and Integration of Business for the Web." [Online]. Available: http://www.uddi.org

[12] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer, "Simple Object Access Protocol," Tech. Rep. NOTE-SOAP-20000508, May 2000. [Online]. Available: http://www.w3.org/TR/SOAP

[13] "Jsr 168: Portlet specification," Web page, Oct 2003. [Online]. Available: http://jcp.org/en/jsr/detail?id=168

[14] "Web services for remote portlets," Web page, Jan 2005. [Online]. Available: http://www.oasis-open.org/committees/download.php/21178/wsrp-primer-1.0.html

[15] "Apache pluto." [Online]. Available: http://portals.apache.org/pluto/

[16] "Apache wsrp4j." [Online]. Available: http://portals.apache.org/wsrp4j/

[17] "uPoartal." [Online]. Available: http://www.uportal.org/

[18] "Openportal." [Online]. Available: https://portal.dev.java.net/

[19] R. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, UNIVERSITY OF CALIFORNIA, IRVINE, 2000. [Online]. Available: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[20] "The atom syndication format," Web page, Dec 2005. [Online]. Available: http://www.ietf.org/rfc/rfc4287

[21] "Rss 2.0 specification," Web page, Oct 2007. [Online]. Available: http://www.rssboard.org/rss-specification

[22] "Extensible markup language (xml)," Web page. [Online]. Available: http://www.w3.org/XML/

[23] "The extensible hypertext markup language," Web page, Aug 2002. [Online]. Available: http://www.w3.org/TR/xhtml1/

[24] "Cascading style sheets," Web page. [Online]. Available: http://www.w3.org/Style/CSS/

[25] ECMA, "Standard ecma-262 ECMAscript language specification, 3rd edition," Dec 1999. [Online]. Available: http://www.ecma-international.org/publications/standards/Ecma-262.htm

[26] W3C, "Simple Object Access Protocol (SOAP) version 1.1," May 2000. [Online]. Available: http://www.w3.org/TR/soap/

[27] IETF, "HTTP State Management Mechanism," Feb 1997. [Online]. Available: http://www.w3.org/Protocols/rfc2109/rfc2109

[28] OASIS, "Web Services Security v1.0 (WS-Security 2004)," 2004. [Online]. Available: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf

[29] G. von Laszewski, I. Foster, J. Gawor, and P. Lane, "A Java Commodity Grid Kit," *Concurrency and Computation: Practice and Experience*, vol. 13, no. 8-9, pp. 643–662, 2001.

[30] S. Microsystems, "Introduction to JSR 168 - The Java Portlet Specification," Web page. [Online]. Available: http://developers.sun.com/portalserver/reference/techart/jsr168/pb_whitepaper.pdf

[31] "Open Grid Computing Environments." [Online]. Available: http://www.ogce.org

[32] J. Novotny, M. Russell, and O. Wehrens, "Gridsphere: an advanced portal framework," in *Euromicro Conference, 2004. Proceedings. 30th*, 2004, pp. 412–419.

[33] "TeraGrid Portal." [Online]. Available: http://www.teragrid.org/userinfo/portal.php

[34] "Condor: High Throughput Computing." [Online]. Available: http://www.cs.wisc.edu/condor/

[35] "Unicore." [Online]. Available: http://www.unicore.de/

[36] "Java CoG Kit," Web Pages. [Online]. Available: http://wiki.cogkit.org/index.php/Java_CoG_Kit

[37] "Gridway metascheduler." [Online]. Available: http://www.gridway.org/doku.php

[38] "Teragrid portal." [Online]. Available: http://teragrid.org/

[39] "Gridsphere portal framework." [Online]. Available: http://www.gridsphere.org/gridsphere/gridsphere

[40] "Same Origin Policy for JavaScript." [Online]. Available: http://developer.mozilla.org/En/Same_origin_policy_for_JavaScript

[41] K. Amin, M. Hategan, G. von Laszewski, and N. J. Zaluzec, "Abstracting the Grid," in *Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2004)*, La Coruña, Spain, 11-13 Feb. 2004, pp. 250–257.

[42] G. von Laszewski, "Java CoG Kit Workflow Concepts," *Journal of Grid Computing*, Jan. 2006, http://dx.doi.org/10.1007/s10723-005-9013-5.

[43] IETF, "The application/json Media Type for JavaScript Object Notation (JSON)," Jul 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4627.txt

[44] "Introducing JSON," Web Page, 2009. [Online]. Available: http://www.json.org/

[45] J. Walker, "JSON is not as safe as people think it is," March 2007. [Online]. Available: http://directwebremoting.org/blog/joe/2007/03/05/json_is_not_as_safe_as_people_think_it_is.html

[46] J. Grossman, "Advanced Web Attack Techniques using GMail," Jan 2006. [Online]. Available: http://jeremiahgrossman.blogspot.com/2006/01/advanced-web-attack-techniques-using.html

[47] R. Yates, "Safe JSON," March 2007. [Online]. Available: http://robubu.com/?p=24

[48] G. Di Lucca, A. Fasolino, M. Mastoianni, and P. Tramontana, "Identifying cross site scripting vulnerabilities in Web applications," in *Web Site Evolution, 2004. WSE 2004. Proceedings. Sixth IEEE International Workshop on*, 11 Sept. 2004, pp. 71–80.

[49] J. Shanmugam and M. Ponnavaikko, "A solution to block Cross Site Scripting Vulnerabilities based on Service Oriented Architecture," in *Computer and Information Science, 2007. ICIS 2007. 6th IEEE/ACIS International Conference on*, 11-13 July 2007, pp. 861–866.

[50] N. Jovanovic, E. Kirda, and C. Kruegel, "Preventing Cross Site Request Forgery Attacks," in *Securecomm and Workshops, 2006*, Aug. 28 2006-Sept. 1 2006, pp. 1–10.

[51] O. Ismail, M. Etoh, Y. Kadobayashi, and S. Yamaguchi, "A Proposal and Implementation of Automatic Detection/Collection System for Cross-site Scripting Vulnerability," in *Proc. 18th International Conference on Advanced Information Networking and Applications AINA 2004*, vol. 1, 2004, pp. 145–151 Vol.1.

[52] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic, "Noxes: a Client-side Solution for Mitigating Cross-site Scripting Attacks," in *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing.* New York, NY, USA: ACM, 2006, pp. 330–337.

[53] "The TLS Protocol Ver 1.0." [Online]. Available: http://tools.ietf.org/html/rfc2246

[54] "Grid Security Infrastructure." [Online]. Available: http://www.globus.org/security/

[55] "Maven." [Online]. Available: http://maven.apache.org/

[56] NCSA, "MyProxyLogon." [Online]. Available: http://grid.ncsa.uiuc.edu/myproxy/MyProxyLogon/

[57] "Apache cxf: An open source service frameworks," webpage. [Online]. Available: http://cxf.apache.org/

[58] S. Microsystems, "Java API for XML Web Services (JAX-WS)." [Online]. Available: https://jax-ws.dev.java.net/

[59] Apache, "Apache Tmocat." [Online]. Available: http://tomcat.apache.org/

[60] "XMLHttpRequest," Web page. [Online]. Available: http://www.w3.org/TR/XMLHttpRequest/

[61] "jQuery." [Online]. Available: http://jquery.com/

[62] "The Dojo Toolkit." [Online]. Available: http://dojotoolkit.org/

[63] "EXT JS Library," Web page. [Online]. Available: http://www.extjs.com/

[64] "Opensocial," Web page. [Online]. Available: http://www.opensocial.org/

[65] "Futuregrid." [Online]. Available: http://futuregrid.org/

A Appendix I List of Deliverables

Please also refer to the project website at:

`http://cyberaide.org/projects/cyberaide/cyberaide-javascript`

for the links to the deliverables.

- Source files and resources, including Java, JavaScript, HTML, CSS, configuration files and some image files.

  These could be found at:

  `http://cyberaide.googlecode.com/svn/trunk/project/javascript/`

  Mediator service source:

```
|-- pom.xml
`-- src
    `-- main
        |-- java
        |   `-- org
        |       `-- cyberaide
        |           |-- account
        |           |   |-- IUserAccount.java
        |           |   |-- UserAccountFile.java
        |           |   |-- UserAccountMem.java
        |           |   |-- UserAccountUtil.java
        |           |   `-- package.html
        |           |-- execution
        |           |   |-- Executable.java
        |           |   |-- FileTransfer.java
        |           |   |-- RemoteJob.java
        |           |   |-- WfKarajan.java
        |           |   |-- WfStatus.java
```

```
|           |   |-- WfType.java
|           |   |-- Workflow.java
|           |   `-- package.html
|           |-- security
|           |   |-- Myproxy.java
|           |   `-- package.html
|           `-- ws
|               `-- mediator
|                   |-- CogMediator.java
|                   |-- CogQueue.java
|                   |-- IMediator.java
|                   |-- MediatorService.java
|                   |-- MediatorServiceMsgMon.java
|                   |-- MediatorServicePasswdCallback.java
|                   `-- package.html
|-- resources
|   `-- log4j.properties
`-- wsdl

12 directories, 24 files
```

Agent service source:

```
|-- pom.xml.in
|-- src
|   `-- main
|       |-- java
|       |   `-- org
|       |       `-- cyberaide
|       |           |-- package.html
|       |           `-- ws
|       |               `-- agent
```

```
|       |                          |-- Agent.java
|       |                          |-- AgentPasswdCallback.java
|       |                          |-- AgentService.java
|       |                          `-- package.html
|       |-- resources
|       |   `-- log4j.properties
|       |-- webapp
|       |   `-- WEB-INF
|       |       |-- cxf-servlet.xml
|       |       |-- web.xml
|       |       `-- wsdl
|       `-- wsdl
`-- tomcat-users.xml.in

12 directories, 11 files
```

Client side source and resources(Images omitted due to space limit):

```
|-- CogKit2.js
|-- cyberaide-0.3.js
|-- cyberaideFiletransfer.xml
|-- cyberaideJobman.xml
|-- cyberaidegadget.js
|-- iepngfix.htc
|-- images
        ......
        ......
|-- jsportal.html
|-- lib
|   |-- dojo.tar.gz
|   |-- interface.js
|   |-- jquery.js
```

```
|   |-- prototype-1.6.0.2.js
|   |-- soapclient24.js
|   |-- soapclient24NSMod.js
|   `-- soapclientGadget.js
|-- pom.xml
`-- tginfo.csv

9 directories, 153 files
```

- Setup script and configuration files:

```
|-- Doxyfile
|-- Makefile
|-- README.txt
|-- certificates.tar.gz
|-- cog.properties.in
|-- cogcerts.tar.gz
|-- install.sh
|-- setenv.sh.in
|-- shutdownmediator.sh
|-- shutdowntomcat.sh
|-- startupmediator.sh
|-- startuptomcat.sh
|-- tools
`-- users.cyberaide
```

- Documentation and Cyberaide JavaScript API coding examples. These include Administrator's manual, User's manual for the Teragrid portal, and Developers manual for the Cyberaide JavaScript API users (All these are also included as seperate sections of the Appendices. HTML version JavaDoc style documents for all the Java and JavaScript source files are available online at the project page.

B Appendix II Administrator's Manual

- This document will guide you through the system setup.
- Contents are subject to change since this is a active project. Please refer to the online project page for the latest document.
- Linux system and Mac OS X system are supported and tested.
- This project requires Java SE6 Update 4 or later.
- The project is managed using Apache Maven2. You must have maven installed to finish the installation.
- The first time when you build the project, a large amount of dependent jar files will be downloaded.

A One-step installation script has been provided to ease the system setup. The script will check all the prerequisite and download the necessary tools and components from the Internet, and then build and deploy the system. After the setup, you will get the service stack and the web application served in the machine where you did the installation.

To start the one-step installation, simply execute the 'install.sh' script in Linux or Mac OS X command line and follow the instructions.

To get more customization, for example, to deploy the Mediator Service and the Web application server into different machines, please follow the instructions below:

B.1 MEDIATOR SERVER INSTALATION

B.1.1 Prerequisite

- Java CoG Kit is needed. Please visit

http://wiki.cogkit.org/wiki/Main_Page

for more information. CoG Kit provided GUI for setup, it should be fairly easy to do so. Make sure the environment variable COG_INSTALL_PATH is set after installation.

- Please take notice on system and network configurations such as GridFTP port. Make sure Globus and CoG Kit is running well.

B.1.2 Service build and deployment

The server is by default installed on port 8998. However, if you like to change it, you can do so in the Makefile. We have conveniently included a PORT variable that you can set.

- To compile use

```
make mediator-build
```

- To deploy the web service use

```
make mediator-run
```

Calling make mediator-run will set up the server and the console waits till the server has been interrupted. You can do this with CTRL+c.

- If you like to combine build and run in one command, you can also use

```
make mediator
```

- To see if the server is running, simply check in your browser the following link:

http://lily01.rit.edu:8998/mediator?wsdl

Please replace the host and port part with your actual settings.

You should see the wsdl document describing the server.

B.2 AGENT SERVICE INSTALATION

B.2.1 Prerequisite

- TOMCAT need to be installed. Please refer to

  `http://tomcat.apache.org/`

  for more information.

- Configure HTTPS connection for tomcat. Please see

  `http://tomcat.apache.org/tomcat-6.0-doc/ssl-howto.html`

B.2.2 Service build and deployment

- Use

```
make agent-pom
```

  to generate the appropriate pom file for the agent service

- In order for the agent to be able to be compiled the server must be running. To compile use

```
make agent-build
```

  Then the WAR file called Agent.war would be generated under directory

  `./agent/target`

- Simply put the war file into your TOMCAT web application directory to complete the deployment. The directory is like:

  `$TOMCAT_HOME/webapps/`

B.3 JAVASCRIPT API AND WEB PORTAL INSTALL

B.3.1 Prerequisite

- The same as in Agent service part.
- Currently tested on Firefox 3.0+.

B.3.2 Web application/portal installation

- Download Dojo toolkit 1.2.0 from

  http://dojotoolkit.org/downloads

- Unzip and put the whole directory under the ./client/lib directory. Rename the directory name to 'dojo', in case of version number is included in the name. (We don't include this into our code repository due to the space limit).

- copy all the content of directory ./client into the directory where you want to put your portal. For example, if you put them under

  ```
  $TOMCAT_HOME/webapps/grid/
  ```

  Then the URL of the portal page should be like:

  ```
  https://lily01.rit.edu:8443/grid/jsportal.html
  ```

- Now open this link to test whether the portal page is installed correctly. Then try to authenticate using MyProxy. If you can logged in successfully, congratulations! You have finished the installation of all the tiers and the system is running.

C Appendix III Portal User's Manual

- This manual provides guides on how to use the JavaScript Grid portal using screenshots.
- Contents are subject to change since this is a active project. Please refer to the online project page for the latest document.

C.1 Introduction

The JavaScript Grid Portal is trying to find a way to access Grid through Web browser, while using Web 2.0 technologies.

- The portal provides Mac OS X style UI as in Figure 27

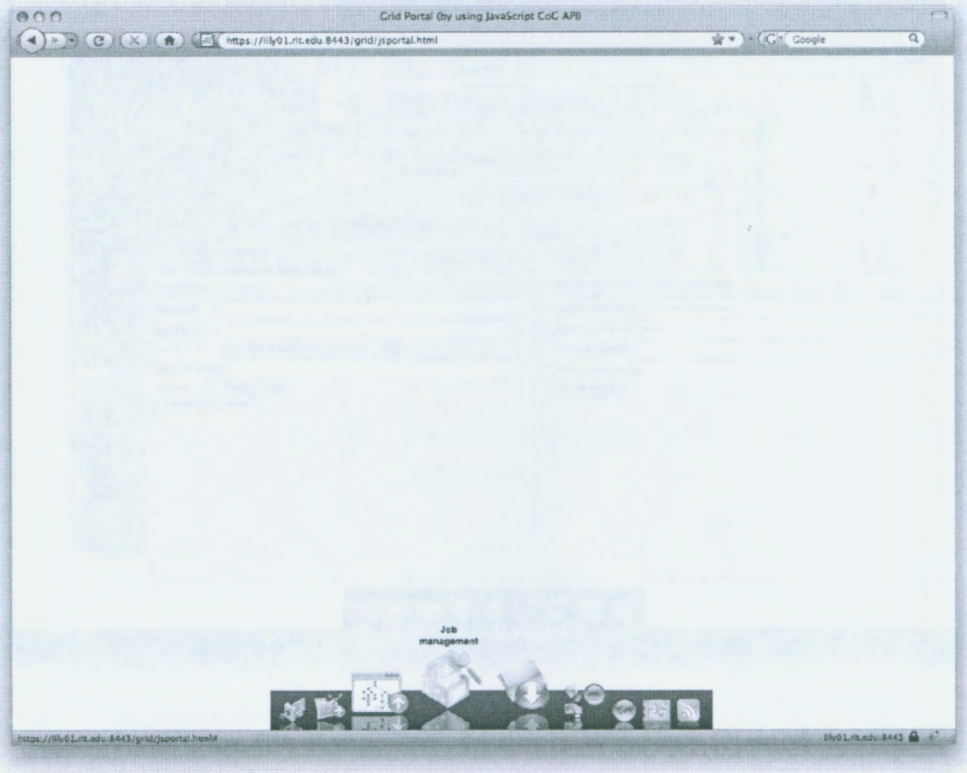


Figure 27: Portal UI in Mac OS X Style

- Each icon is corresponding to one functionality, from authentication, job submission, status monitor and result retrieve to file transfer and information services, shown in Figure 28.

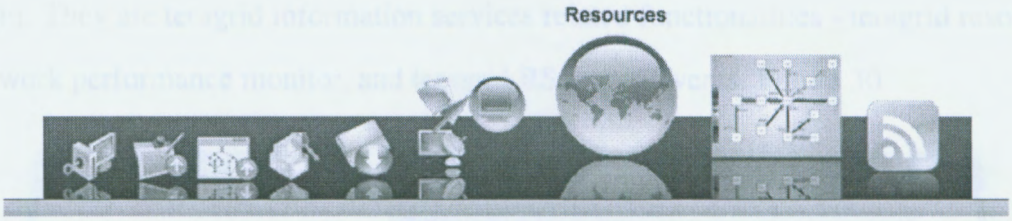


Figure 28: Functionalities and Icons

- When all windows are opened and cascaded, Figure 29.

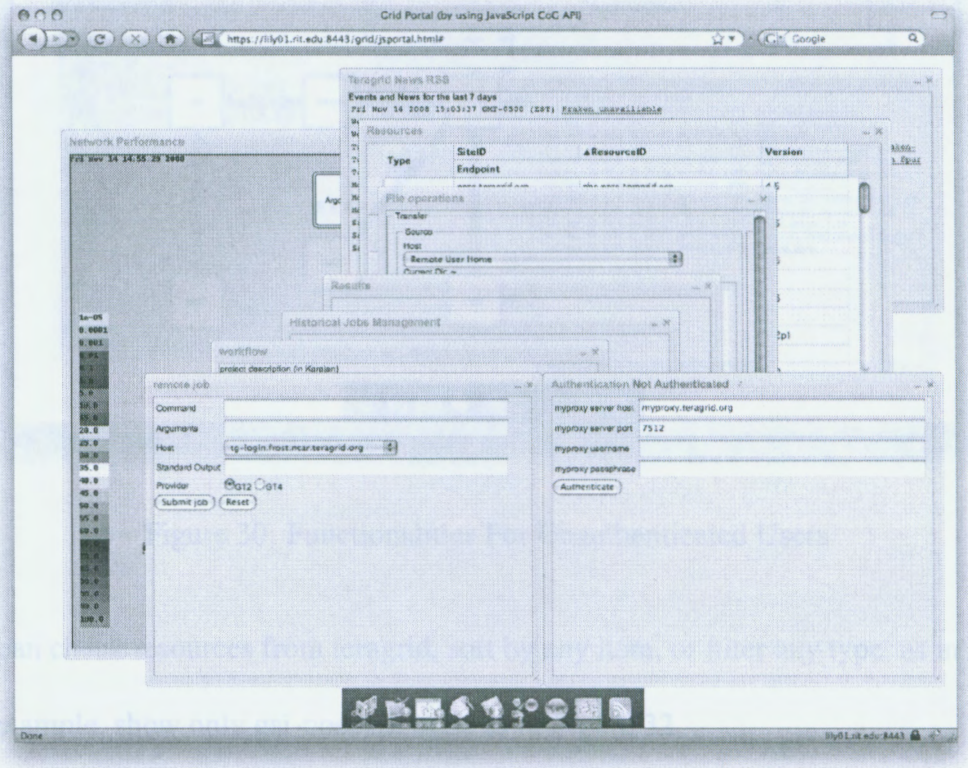


Figure 29: Portal With Multiple Windows Opened

C.2 First Look: Teragrid Information Services

- There are several functionalities that support anonymous access, i.e., user does not need to login. They are teragrid information services related functionalities - teragrid resources list, network performance monitor, and teragrid RSS news/events. Figure 30

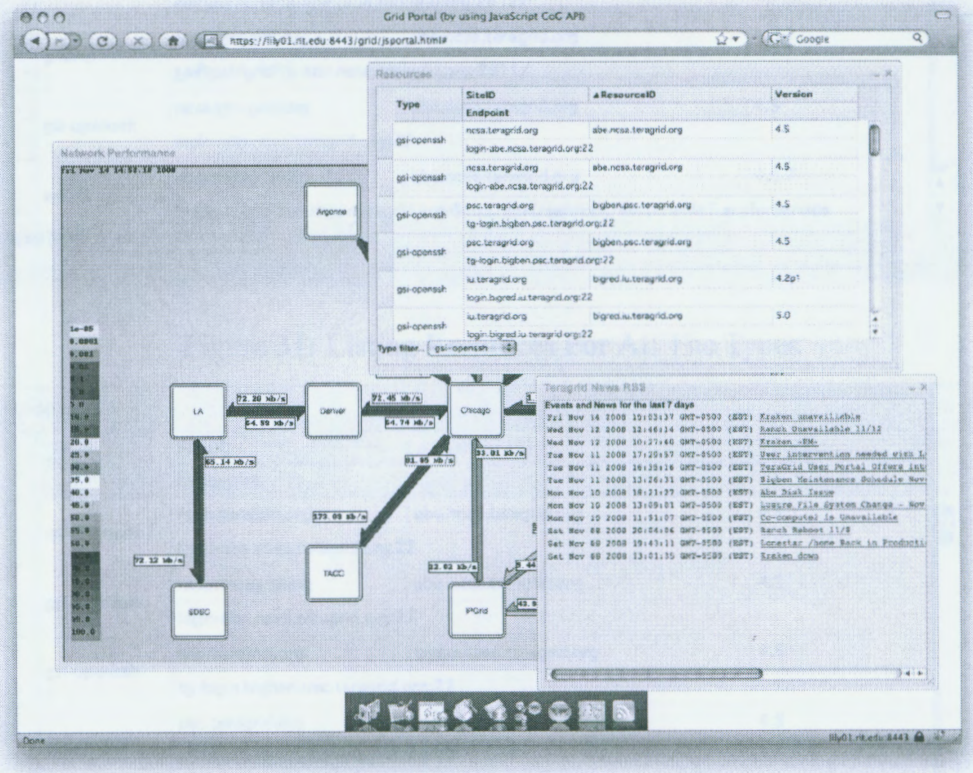


Figure 30: Functionalities For Unauthenticated Users

- You can check resources from teragrid, sort by any item, or filter any type, as in Figure 31. For example, show only gsi-openssh node as in Figure 32.

- A semi realtime network performance map is presented as shown in Figure 33.

- Teragrid RSS news/events reader is shown in Figure 34. Click topic will pop up a new window to display detailed information for that topic

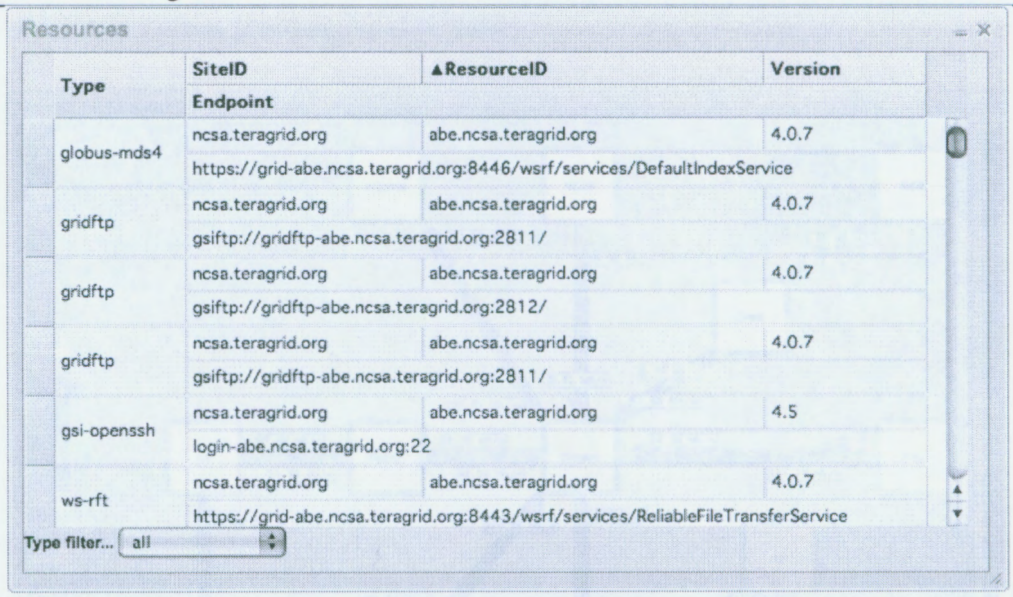


Figure 31: Listing Resources For All The Types

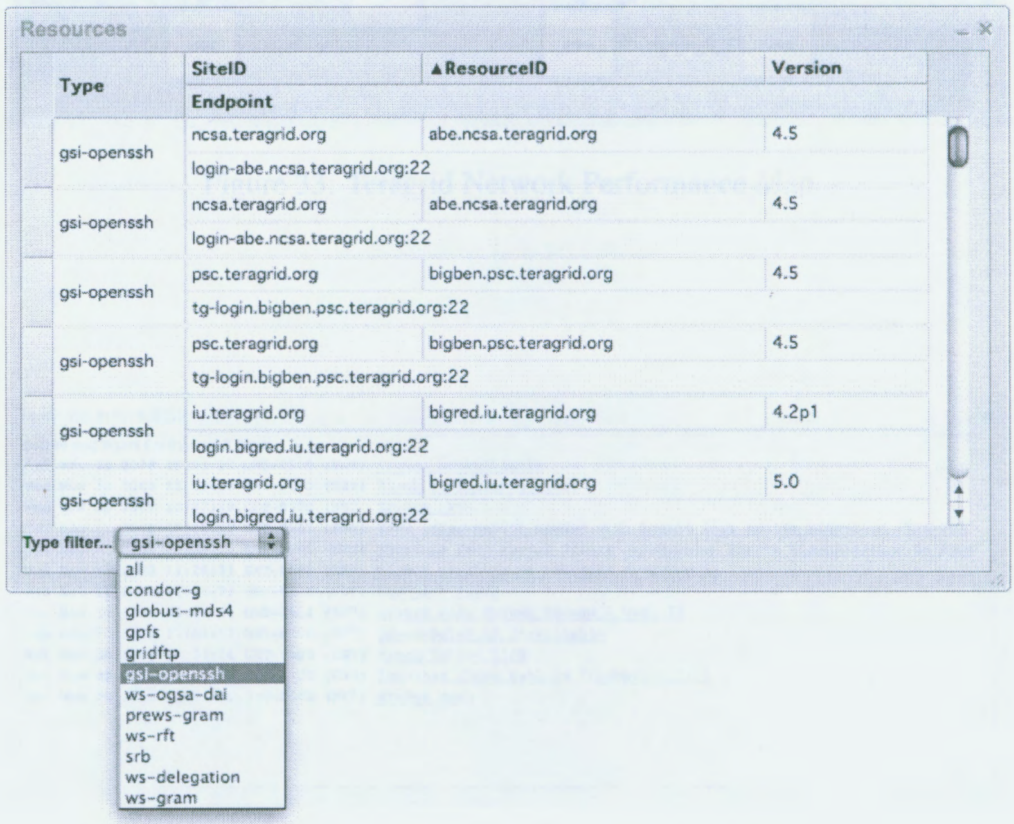


Figure 32: Resources Type Filtered

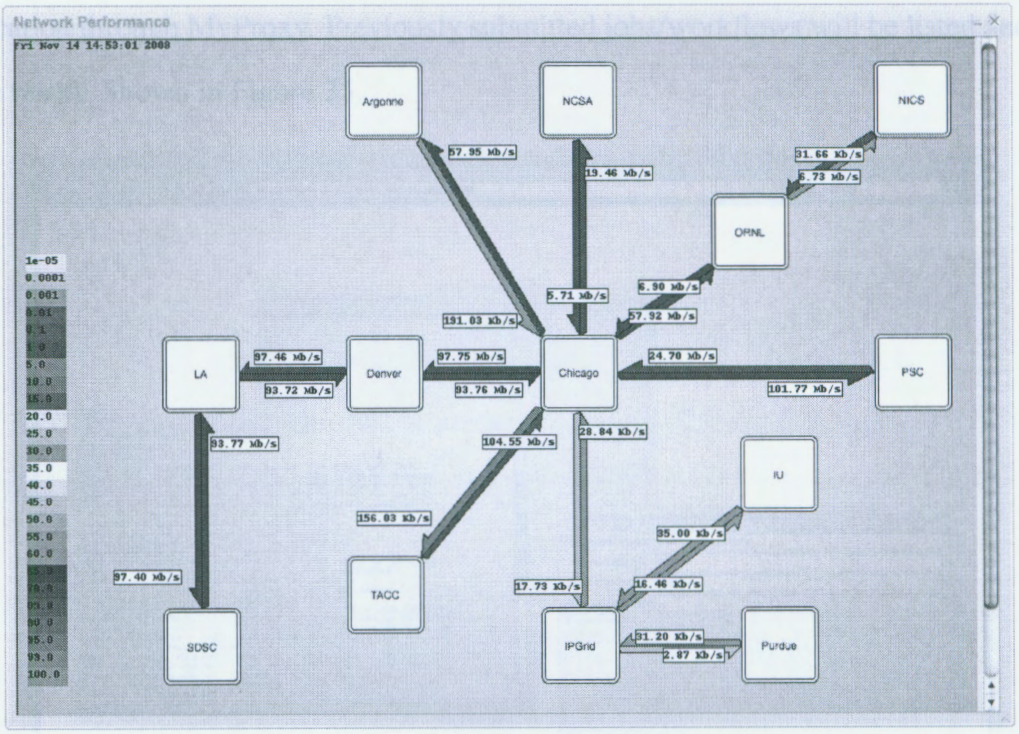


Figure 33: Teragrid Network Performance Map

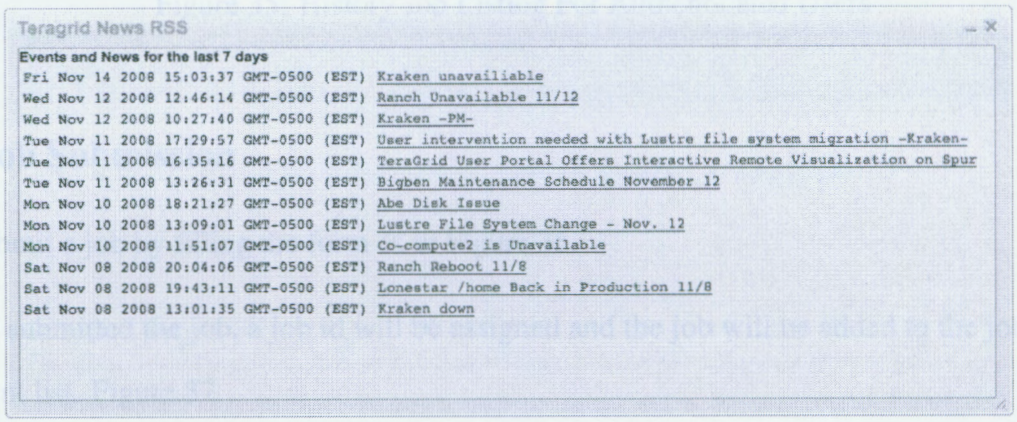


Figure 34: Teragrid News/Events RSS Feeds

C.3 Authentication through MyProxy

Authentication through MyProxy. Previously submitted jobs/workflows will be listed and user can check the result. Shown in Figure 35.

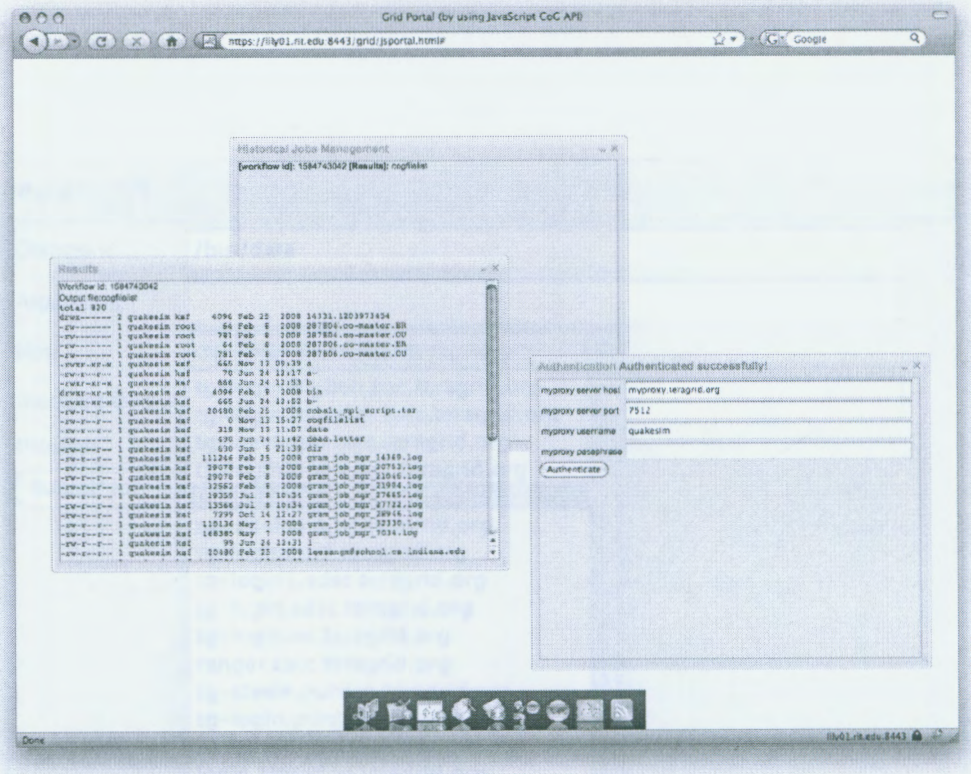


Figure 35: History Job Listing For Authenticated Users

C.4 Job Submission

- Submit a job by filling the form as in Figure 36.
- By submitted the job, a job id will be assigned and the job will be added to the job management list. Figure 37.
- By clicking the jobid user could monitor the execution status, and upon finishing the result could be displayed. Figure 38.

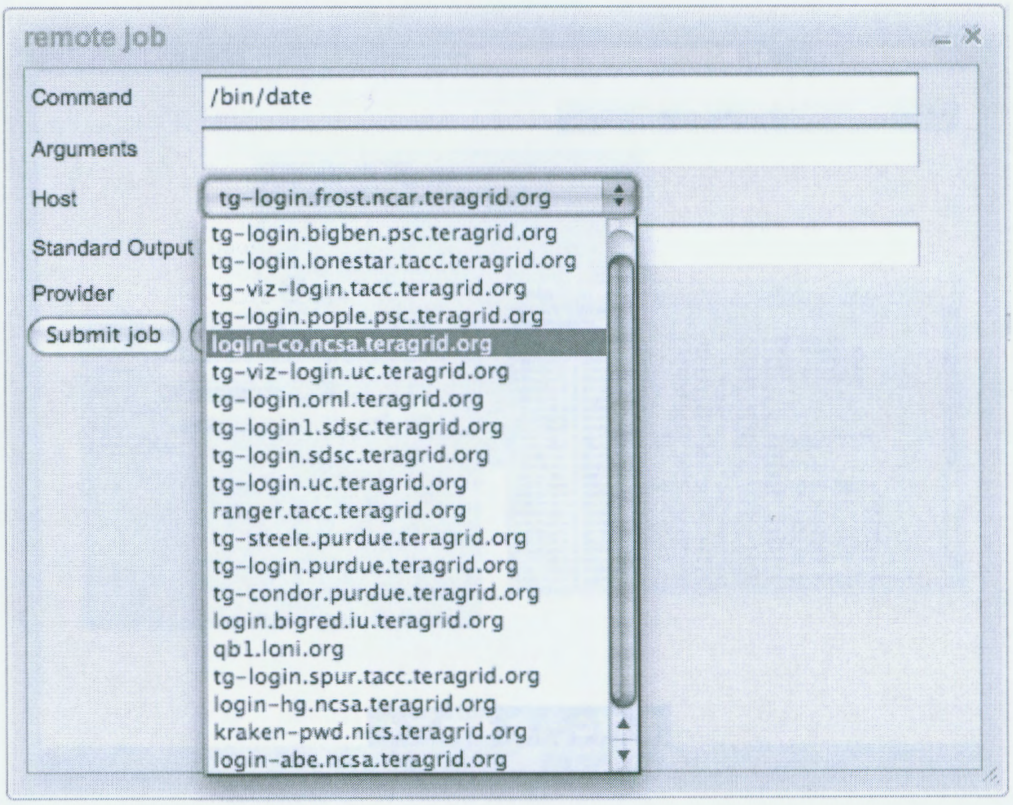


Figure 36: Submitting A Job To Available Resources

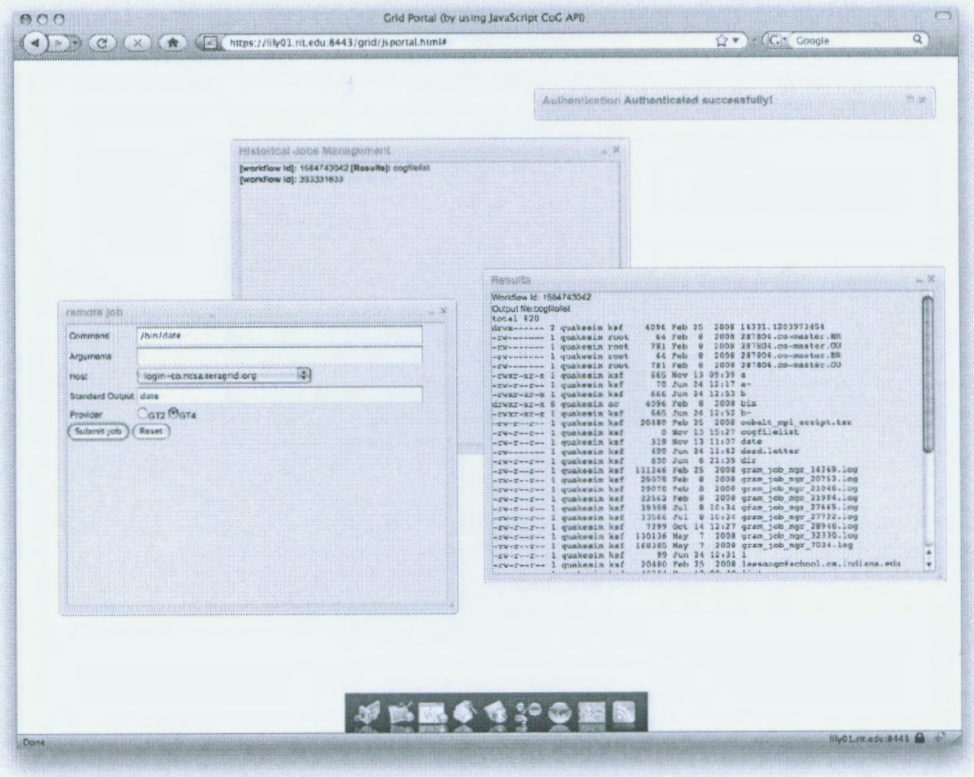


Figure 37: Job Ids Shown For Status Query And Result Retrieval

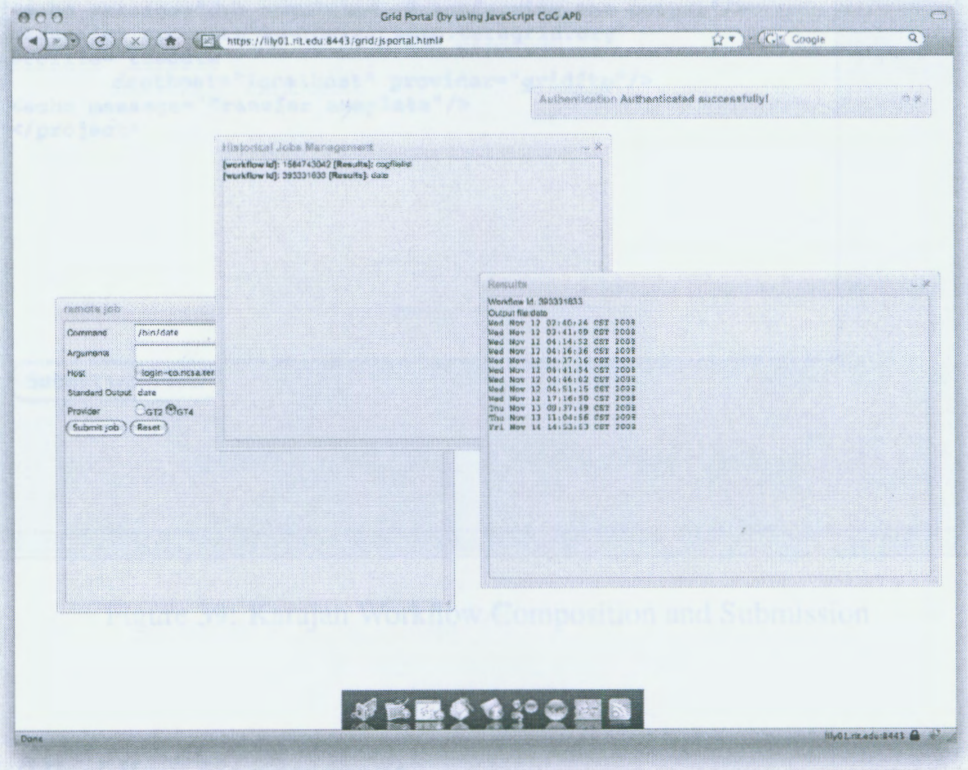


Figure 38: Retrieved Job Execution Result Shown

C.5 Workflow Submission (by using Karajan)

You can submit a workflow described by the Karajan workflow language. See Figure 39.

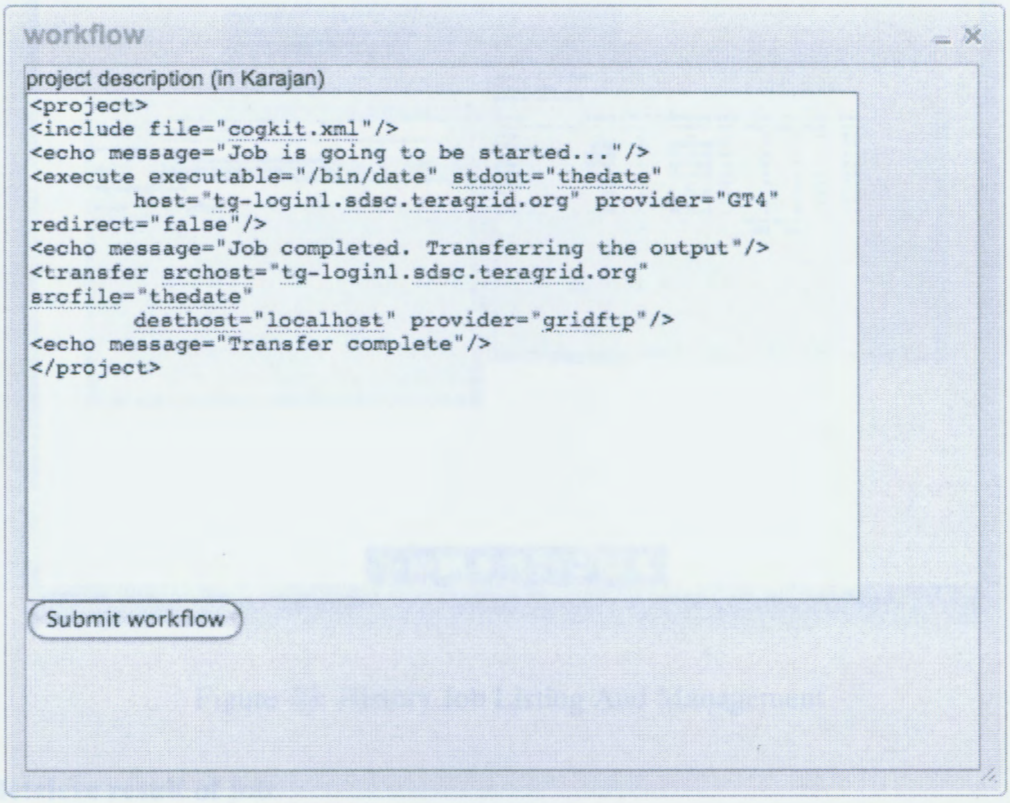


Figure 39: Karajan Workflow Composition and Submission

C.6 Job/Workflow management

Submit window, job management window and result window are displayed while the user logged in. See Figure 40.

C.6.1 Check status of previously submitted job

Click the jobid will check the job's execution status. See Figure 41.

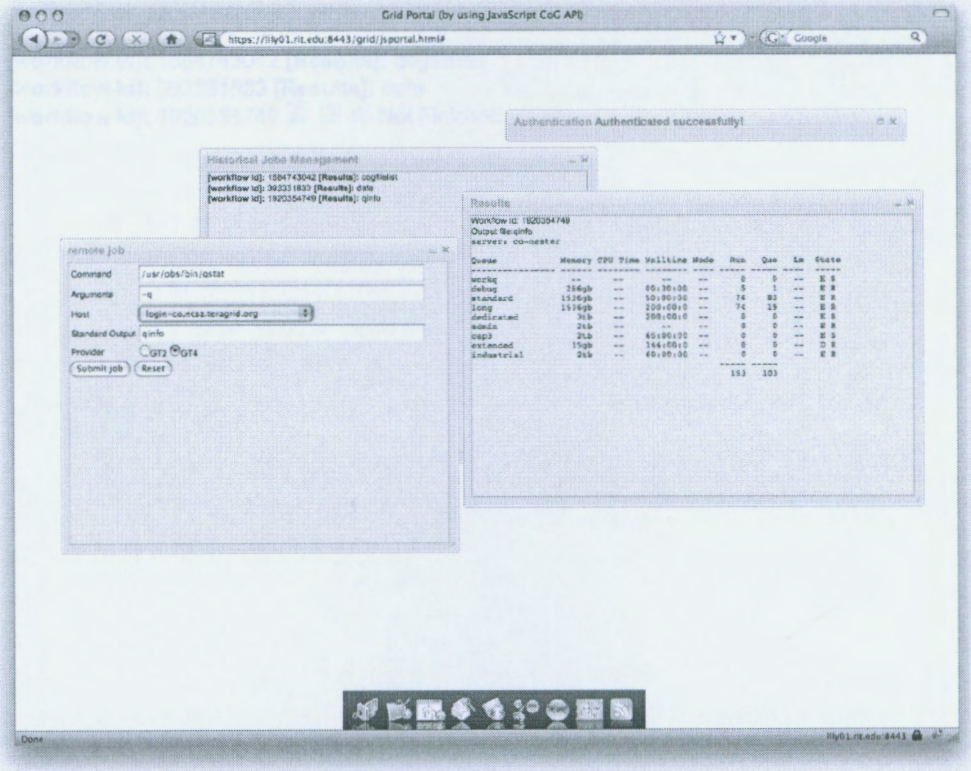


Figure 40: History Job Listing And Management

C.6.2 Retrieve result of job

And upon finishing user could retrieve the result back and display it in the result windows. See Figure 42.

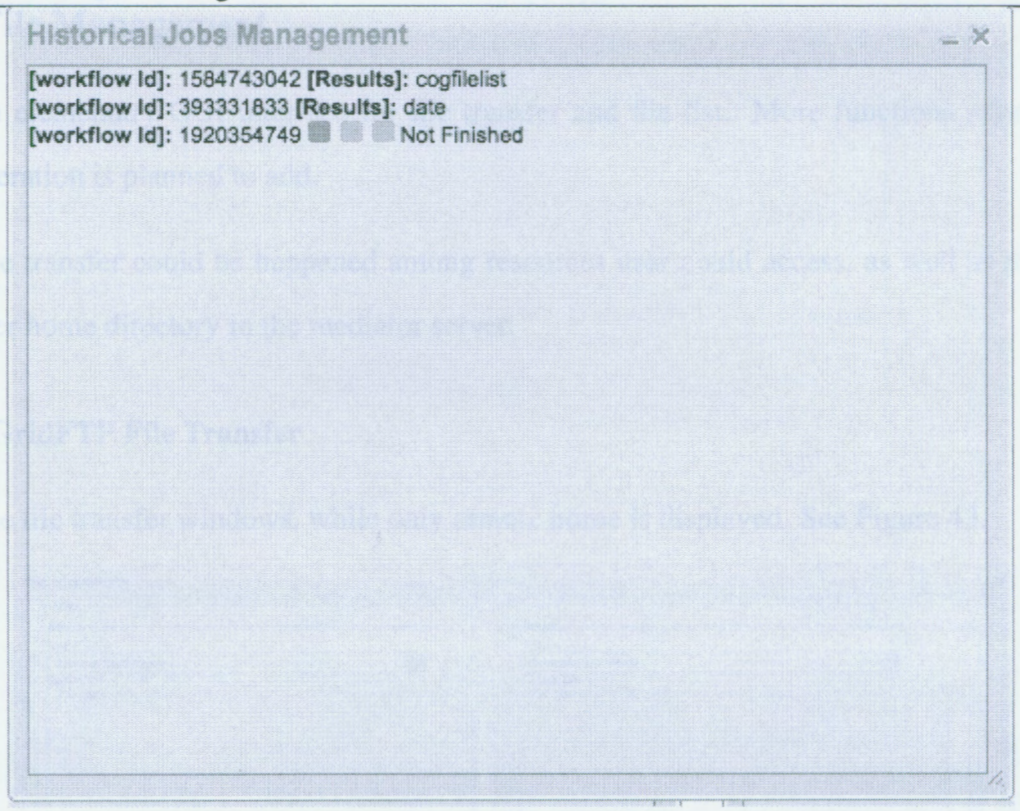


Figure 41: Check Job Execution Status

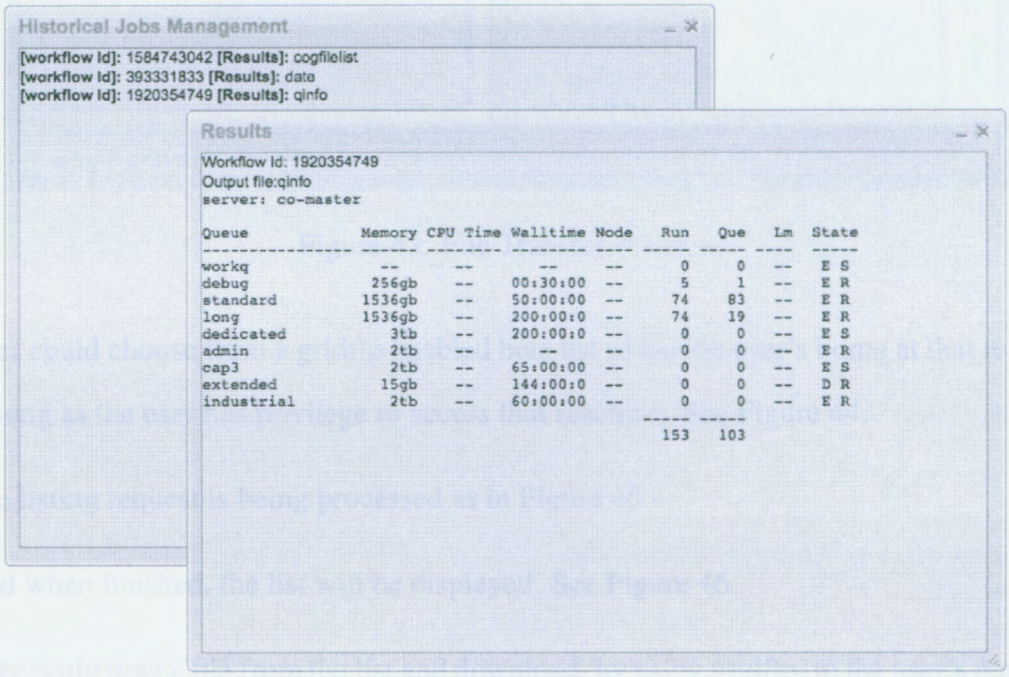


Figure 42: Retrieve And Display Job Execution Result

C.7 File Management

- An elementary GUI interface for file transfer and file list. More functions related to file operation is planned to add.
- File transfer could be happened among resources user could access, as well as the remote user home directory in the mediator server.

C.7.1 GridFTP File Transfer

- The file transfer windows, while only remote home is displayed. See Figure 43.

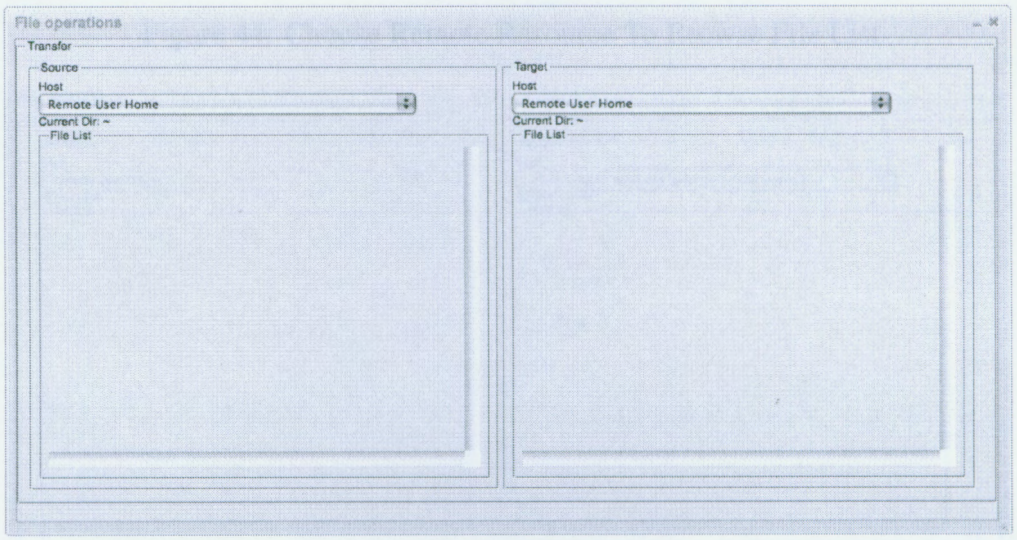


Figure 43: File Transfer Window

- User could choose from a gridftp enabled host list to list the user's home at that remote host, as long as the user has privilege to access that machine. See Figure 44.
- File listing request is being processed as in Figure 45.
- And when finished, the list will be displayed. See Figure 46.
- User could drag a file from the list and download, by using gridftp, to the user's remote home directory in mediator service. See Figure 47.

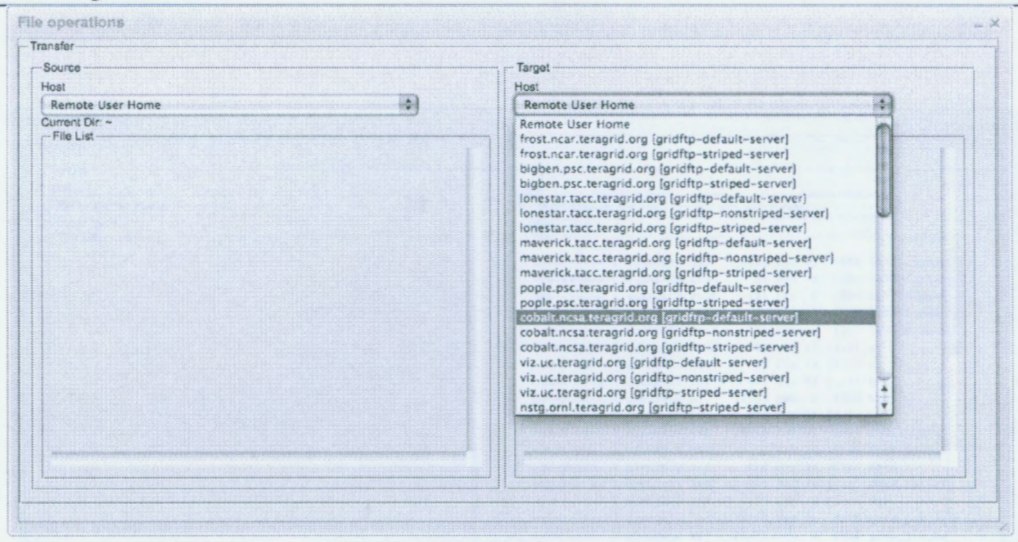


Figure 44: Choose Remote Resource To Browse File List

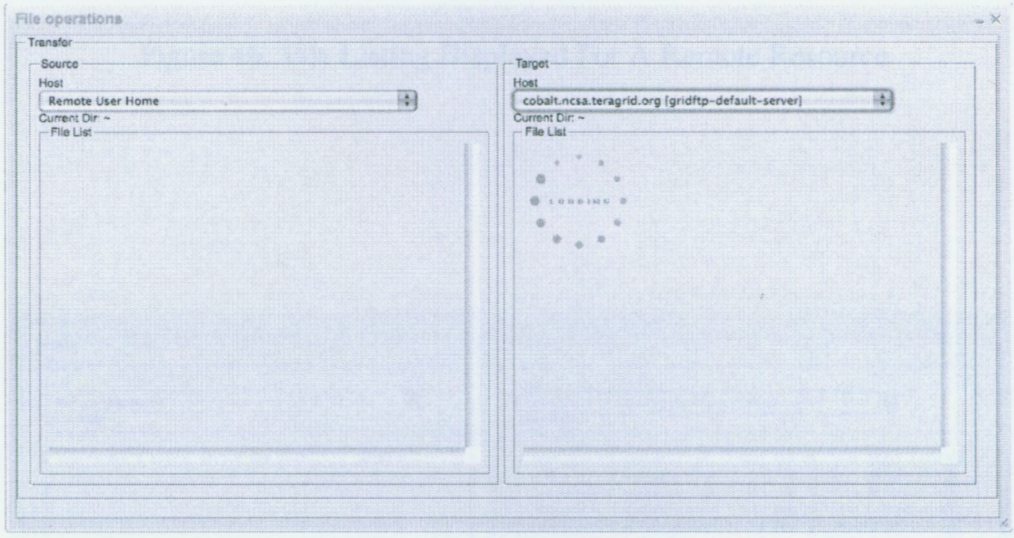


Figure 45: File Listing Being Processed

- Or vice versa, as in Figure 48.
- Or between two resources from teragrid. See Figure 49.

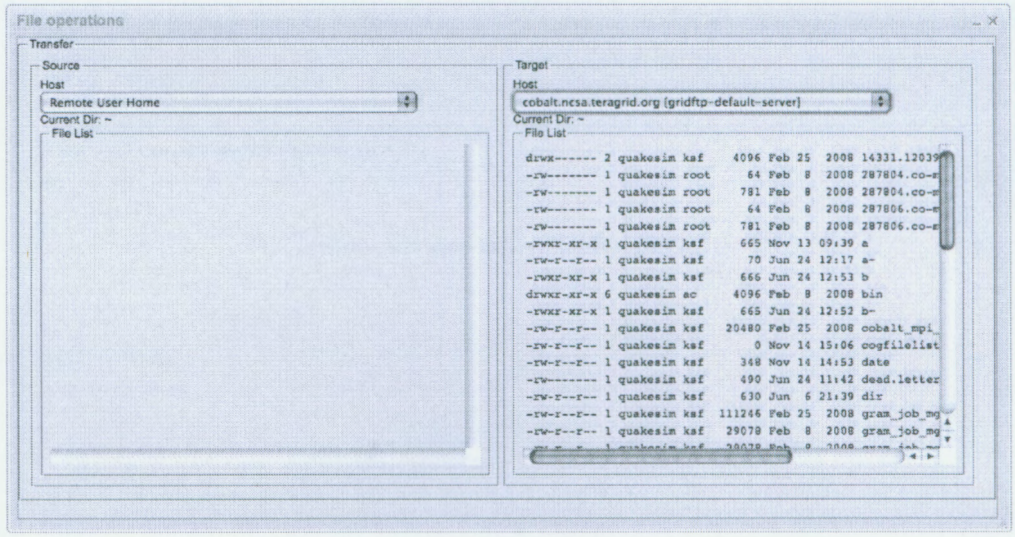


Figure 46: File Listing Displayed For A Remote Resource

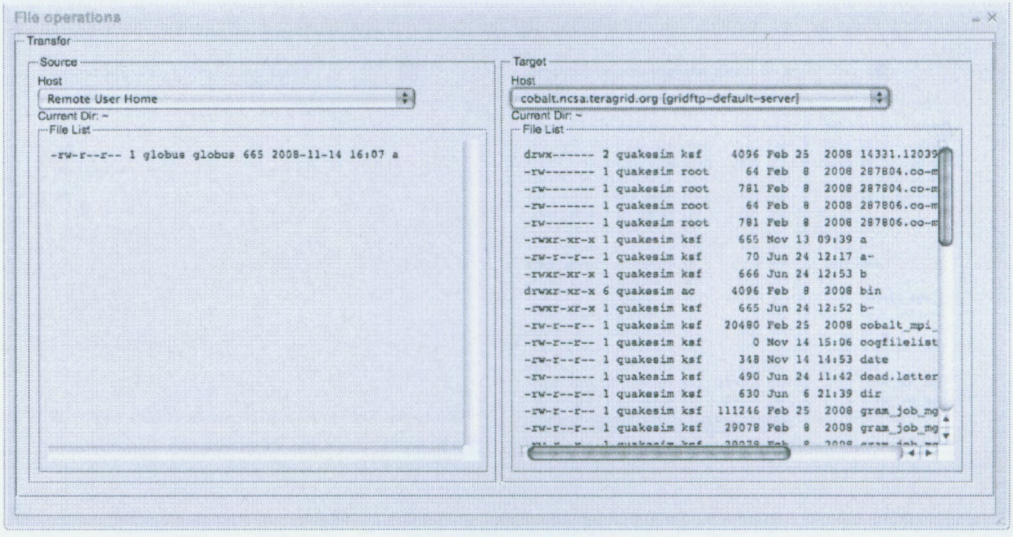


Figure 47: File Transfered From Remote Resource

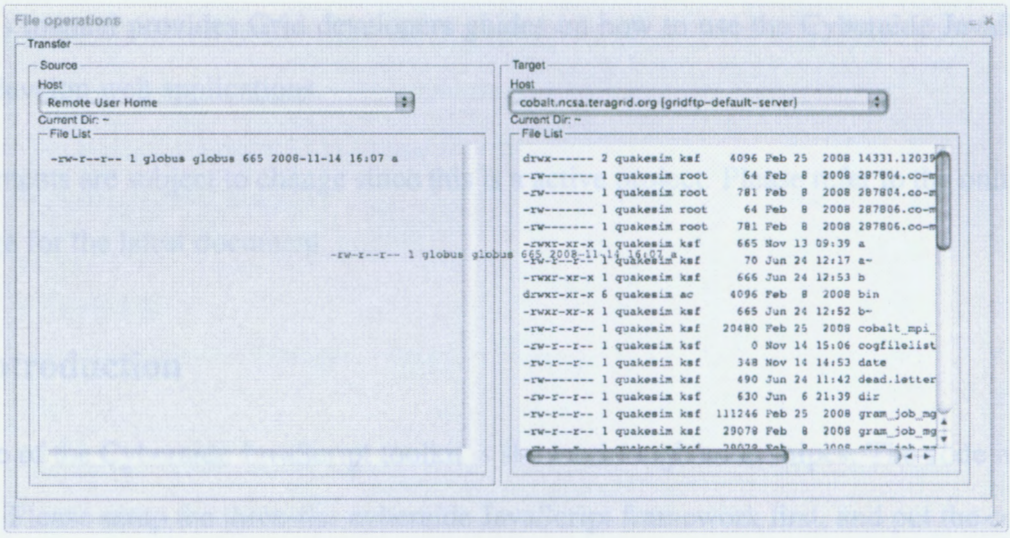


Figure 48: Drag And Drop Style File Transfer

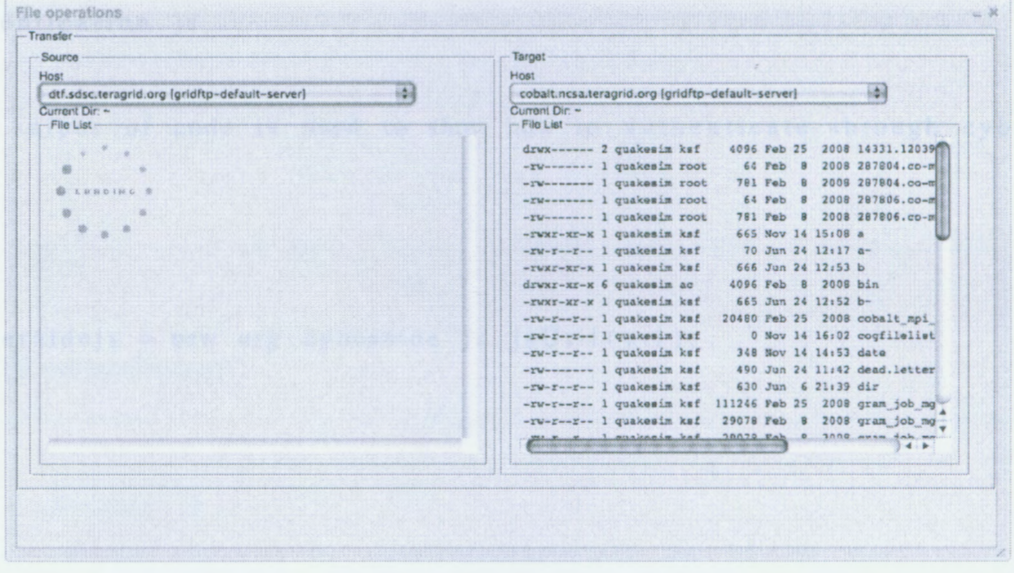


Figure 49: File Being Transfered Between Teragrid Resources

D Appendix IV Developer's Manual

- This manual provides Grid developers guides on how to use the Cyberaide JavaScript API to develop web applications.
- Contents are subject to change since this is a active project. Please refer to the online project page for the latest document.

D.1 Introduction

The usage of the Cyberaide JavaScript toolkit is shown through code snipes. The code itself is not runnable. Please setup the three-tier cyberaide JavaScript framework first, and put the code snipes into your code where you want to fulfill the functionality.

D.2 Authentication

```
/*
 * authentication.js
 *
 * This snipes of code is used to show how to authenticate through cyberaidejs.
 *
 */

var cyberaidejs = new org.cyberaide.js.jsUtil(url);
//
//......
//

//construct authenticator object for authentication using myproxy
//make sure to use the attributes keys specified here.
var auth = org.cyberaide.js.jsAuthenticator(url);
```

```
auth.setAttribute("host", "myproxy.teragrid.org");
auth.setAttribute("port", 7512);
auth.setAttribute("user", 'USER');
auth.setAttribute("password", 'PASSWORD');
//currently only 'myproxy' is supported
auth.setProvider("myproxy");

// authentication
cyberaidejs.authenticate(auth, authResponse);

/*
 * the authenticate callback function
 */
function authResponse(ret) {
    if(ret){
        //authenticated successfully!
        //put your code here
    } else{
        //authentication failed
        //put your code here
    }
}
```

D.3 job submission

```
/*
 * job-submission.js
 *
 * This snippes of code is used to show how to submit a job to
 * remote machine to execute.
 *
```

```
 */

//construct cyberaidejs object by pointing to the agent service's url
var cyberaidejs = new org.cyberaide.js.jsUtil(url);
//
//......
//


//define the job
//make sure to use the attributes keys specified here.
var execObj = new org.cyberaide.js.jsExecutable();
execObj.setAttribute("cmd", "/bin/ls");
execObj.setAttribute("arg", "-l");
execObj.setAttribute("rHost", 'REMOTEHOST');
execObj.setAttribute("stdout", "lsoutput");
execObj.setAttribute("provider", "GT4");

//construct a remote job through the executable object
var strProj = cyberaidejs.constructRemoteJob(execObj);


//submit the job by specifying constructed job specification,
//callback function.
//you must be in authenticated status and in a valid session.
cyberaidejs.submit(strProj, submitResponse);


/*
 * callback function of submission
 */
function submitResponse(ret) {
    if(ret > 0){
        //job submitted and job id returned.
```

```
        //your job id is 'ret', in Number format
        //do something here.

    } else {
        //job submission failed.
        //do something here.
    }
}
```

D.4 workflow submission

```
/*
 * workflow.js
 *
 * This snippes of code is used to show how to submit a workflow
 * (in Karajan format) to run on a remote machine.
 *
 */

//construct cyberaidejs object by pointing to the agent service's url
var cyberaidejs = new org.cyberaide.js.jsUtil(url);
//
//......
//


//
//var strProj = ...;
//assign the Karajan workflow to this string and submit it.
//you must be in authenticated status and in a valid session.
cyberaidejs.submit(strProj, submitResponse, null);
```

```
/*
 * callback function of submission
 */
function submitResponse(ret) {
    if(ret > 0){
        //job submitted and job id returned.
        //your job id is 'ret', in Number format
        //do something here.

    } else {
        //job submission failed.
        //do something here.
    }
}
```

D.5 list submitted jobs for the authenticated user

```
/*
 * listjobs.js
 *
 * This snippes of code is used to show how to query job/workflow
 * list that user submitted.
 *
 */

//construct cyberaidejs object by pointing to the agent service's url
var cyberaidejs = new org.cyberaide.js.jsUtil(url);
//
//......
//
```

```
//do this only after you have been authenticated and in a valid session.
cyberaidejs.list(listResponse);

/*
 * Submitted jobs listing
 */
function listResponse(jsonRet){
    if(jsonRet != null){
        var jsonRetObj = eval("(" + jsonRet + ")");
        var wfids = jsonRetObj.wfids;
        var numIds = wfids.length;
        //now you got the jobs list with 'numIds' items
        //in an array wfids
        //do something here.
    }
}
```

D.6 query job/workflow execution status

```
/*
 * jobstatus.js
 *
 * This snippes of code is used to show how to query job/workflow
 * execution status.
 *
 */

//construct cyberaidejs object by pointing to the agent service's url
var cyberaidejs = new org.cyberaide.js.jsUtil(url);
//
//......
```

```
//

//do this only after you have submitted some job/workflow, and
//then query the status with its id
cyberaidejs.statusQuery(wfid, statusQueryResponse);


/*
 * callback function of status query, to display the status info
 */
function statusQueryResponse(jsonRet){
    if(jsonRet != null){
        var jsonRetObj = eval("(" + jsonRet + ")")
        var wfid = jsonRetObj.wfid;
        var status = jsonRetObj.status;
        //now you get the workflowid and its current status,
        //in a 'Number' format,
        //representing jobs finished so far in the workflow.
        //do something here.
    }
}
```

D.7 query output filenames for a job/workflow execution

```
/*
 * queryoutput.js
 *
 * This snippes of code is used to show how to query output filename
 * of a job/workflow.
 *
 */
```

```
//construct cyberaidejs object by pointing to the agent service's url
var cyberaidejs = new org.cyberaide.js.jsUtil(url);
//
//......
//

//do this only after you have been authenticated and in a valid session.
cyberaidejs.queryOutput(wfid, queryOutputResponse);

/*
 * display the returned output file names appropriately and add
 * content retrieve links
 */
function queryOutputResponse(jsonRet){
    if(jsonRet != null){
        var jsonRetObj = eval("(" + jsonRet + ")");
        var wfid = jsonRetObj.wfid;
        var resultFiles = jsonRetObj.resultFiles;
        //the result output file names for job/workflow with
        //id 'wfid' have been stored
        //into array 'resultFiles'
        //you can display the filenames or get the REAL output
        //through getOutput() method.
    }
    return false;
}
```

D.8 get execution output

```
/*
 * getoutput.js
```

```
 *
 * This snipes of code is used to show how to get a specific output
 * from a job/workflow execution.
 *
 */

//construct cyberaidejs object by pointing to the agent service's url
var cyberaidejs = new org.cyberaide.js.jsUtil(url);
//
//......
//


//do this only after you have been authenticated and in a valid session.
//wfid and filename specified job/workflow id and (one of) its output
//filename obtained through queryOutput
cyberaidejs.getOutput(wfid, filename, getOutputResponse);

/*
 * processing the returned output
 */
function getOutputResponse(jsonRet){
    if(jsonRet != null){
        var jsonRetObj = eval("(" + jsonRet + ")");
        var wfid = jsonRetObj.wfid;
        var filename = jsonRetObj.filename;
        var content = jsonRetObj.content;
        //now you have got the output content of an
        //output file for the job 'wfid'
        //your code for further processing goes herer..
    }
}
```

D.9 file transfer

```
/*
 * filetransfer.js
 *
 * This snipes of code is used to show how to do file
 * transfer through cyberaidejs.
 *
 */

var cyberaidejs = new org.cyberaide.js.jsUtil(url);
//
//......
//

//source and dest are all URI per gridftp supported format,
//like file://PATH, gsiftp://HOST:PORT/PATH/TO/FILE
//when using file:///~, it represents the user's remote home
//directory at where the mediator service resides,
//using gsiftp://HOST:PORT/PATH/TO/FILE to point to the location
//at other remote resources like those from Teragrid
//do this only after you have been successfully authenticated
//and is in valid session
cyberaidejs.transfer(source, dest, transferResponse);

/*
 * the transfer callback function
 */
function transferResponse(ret, updateloc) {
    if(!ret){
        //Transfered successfully!
        //updateloc indicate the parameter you used to call
```

```
        //the transfer function which is 'dest'
        //in this ccase
        //do whatever you want here.
    } else{
        //Transfered failed!
        //updateloc indicate the parameter you used to call
        //the transfer function which is 'dest'
        //in this ccase
        //do whatever you want here.
    }
}
```