Available online at www.sciencedirect.com
ScienceDirect
Physics of the Earth and Planetary Interiors 163 (2007) 333–346
PHYSICS OF THE EARTH AND PLANETARY INTERIORS
www.elsevier.com/locate/pepi

VLAB: Web services, portlets, and workflows for enabling cyber-infrastructure in computational mineral physics

Evan F. Bollig [a], Paul A. Jensen [b], Martin D. Lyness [b], Mehmet A. Nacar [c], Pedro R.C. da Silveira [b], Dan Kigelman [b], Gordon Erlebacher [a,*], Marlon Pierce [c], David A. Yuen [b], Cesar R.S. da Silva [b]

[a] *School of Computational Science, Florida State University, FL 32306-4120, USA*
[b] *Minnesota Supercomputer Institute, University of Minnesota, MN, USA*
[c] *Community Grids Lab, Indiana University, IN, USA*

Received 5 February 2007; accepted 15 March 2007

Abstract

Virtual organizations are rapidly changing the way scientific communities perform research. Web-based portals, environments designed for collaboration and sharing data, have now become the nexus of distributed high performance computing. Within this paper, we address the infrastructure of the Virtual Laboratory for Earth and Planetary Materials (VLab), an organization dedicated to using quantum calculations to solve problems in mineral physics. VLab provides a front-end portal, accessible from a browser, for scientists to submit large-scale simulations and interactively analyze their results. The cyber-infrastructure of VLab concentrates on scientific workflows, portal development, responsive user-interfaces and automatic generation of web services, all necessary to ensure a maximum degree of flexibility and ease of use for both the expert scientist and the layperson.

Keywords: Grid computing; Workflows; Web services; Visualization; Automation

1. Introduction

During the 1950s, materials science and chemistry communities established the foundations of a long-standing history of advances and expertise in the field of high-performance computing. Mineral physics also joined forces in the late 1970s to 1980s after the successful introduction of quantum mechanics into geophysics (Bukowinski, 1976). Since then, high-performance computing has exploded in popularity into an interdisciplinary area utilized by nearly every scientific discipline. In recent years, severe impediments to the successful execution of large-scale projects have emerged: namely, the proliferation of mature community codes, geographically spread teams and relatively scarce leading-edge computational resources. Some computational problems in mineral physics (Cohen, 2005) lie beyond the capacity of the resources that domain scientists use routinely. Rather, they depend on the integration of extremely high computational power and storage capacity. The ability to acquire and manage such resources conveniently is important and collaboration is essential. As a result, attention has been drawn to the need of "cyber-infrastructure" to enable integration of diverse sources of knowledge (Foster et al., 2001).

* Corresponding author. Tel.: +1 850 644 0186.
E-mail address: erlebach@scs.fsu.edu (G. Erlebacher).

0031-9201/$ – see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.pepi.2007.03.005

With the advent of Web-2.0, which allows for more lively modes of interaction and collaboration among researchers, portals – central repositories of information – have emerged as a focal point for sharing information and have begun to act as an abstraction between scientists and the complexities of simulations. Virtual organizations, such as the Virtual Laboratory for Earth and Planetary Materials (VLab, http://www.vlab.msi.umn.edu) at the University of Minnesota, have emerged as a powerful concept in organizing team-efforts and team-based scientific research. The VLab, established in 2004 by an ITR NSF grant, is a dynamic consortium consisting of material, geo and computer scientists, higher-educational institutions in the USA and Europe and high-end computational resources united by a common interest in solving outstanding problems in mineral physics.

Other virtual organizations in the geo-sciences include CIG (http://www.geodynamics.org), GEON (http://www.geongrid.org), and iSERVO (http://servo.jpl.nasa.gov). The goal of CIG is to serve as a depository of standard codes devoted to a variety of computational science areas in geophysics such as mantle convection, geo-dynamo and seismology. The mandate of GEON is to integrate many geological disciplines and share the vast amounts of geological digital data. iSERVO is a computationally-driven virtual organization devoted to earthquakes, inSAR satellite missions, and tsunami hazards, which had its origin from an international virtual organization in earthquake physics (ACES, http://www.quakes.uq.edu.au/ACES). Finally, the Southern California Earthquake Center (SCEC) has provided leadership in large-scale numerical simulation through Grid-computing and the creation of the infrastructure needed for enabling system-level science in earthquake hazards (Jordan and Maechling, 2003).

Experiments of interior planetary conditions are extremely challenging. Ab initio quantum mechanical calculations may provide the only way to obtain key information on materials properties at relevant temperature and pressure conditions inside planetary interiors. The aims of the VLab are three-fold:

1. address materials science problems related to planetary physics;
2. use available, and develop novel, algorithms to address these problems;
3. harness emergent information technologies to construct a new suite of interfaces between the researchers and the software and hardware resources to improve and facilitate existing workflows.

One inevitable consequence of team-oriented investigation is the need to share information and collaborate. Team members might take part in more than one aspect of a particular investigation or a given member might collaborate on several projects. Any given projects might involve input file generation, code submission, data analysis, and visualization. These tasks are often linked and have dependencies among them. A need thus arises for a workflow and associated software, such as a workflow manager, workflow generators, workflow editors, and workflow storage. Workflows might themselves be composed of simpler workflows. An example that illustrates the visualization of elasticity calculations is illustrated in Fig. 1. The generic task of calculating elastic constants of geo-materials directly using quantum mechanics and then interrogating the results in various formats, whether columns of numbers or visualizations of molecular structures, can be expressed as a web of interconnected modules. The user interacts with these workflows through a web-based portal. Through this portal, the user accesses a range of resources related to computation, visualization and analysis. Furthermore, the portal acts as a front end to various software components responsible for tracking the various user activities.

The paper is organized as follows. In Section 2 we discuss the concept of portals and portlets, and their use in VLab. In Section 3, the current state of web interfaces is discussed with an emphasis on static versus dynamic content generators. Section 4 presents workflows and their usefulness in computational mineral physics. Section 5 returns to workflows and discusses automatic web-service generation and its role within VLab. Finally, we conclude on the importance of cyber-infrastructure for collaborative computational environments and present our perspectives for future endeavors.

2. A survey of science portal technology

A science portal (also sometimes called a science gateway) supports the work of a scientific team or community by combining a web portal and associated web Services. By using a web browser, a scientist can access both private and shared workspaces of discipline-specific data and tools. Science gateways are also access points to Grids (Berman et al., 2003; Foster and Kesselman, 1999; Foster and Kesselman, 2004) of computational and data resources, allowing a user to leverage the capability of a Grid without knowledge of its underlying complexities. These gateways add value to the core Grid infrastructure by providing the following important capabilities and enhancements:

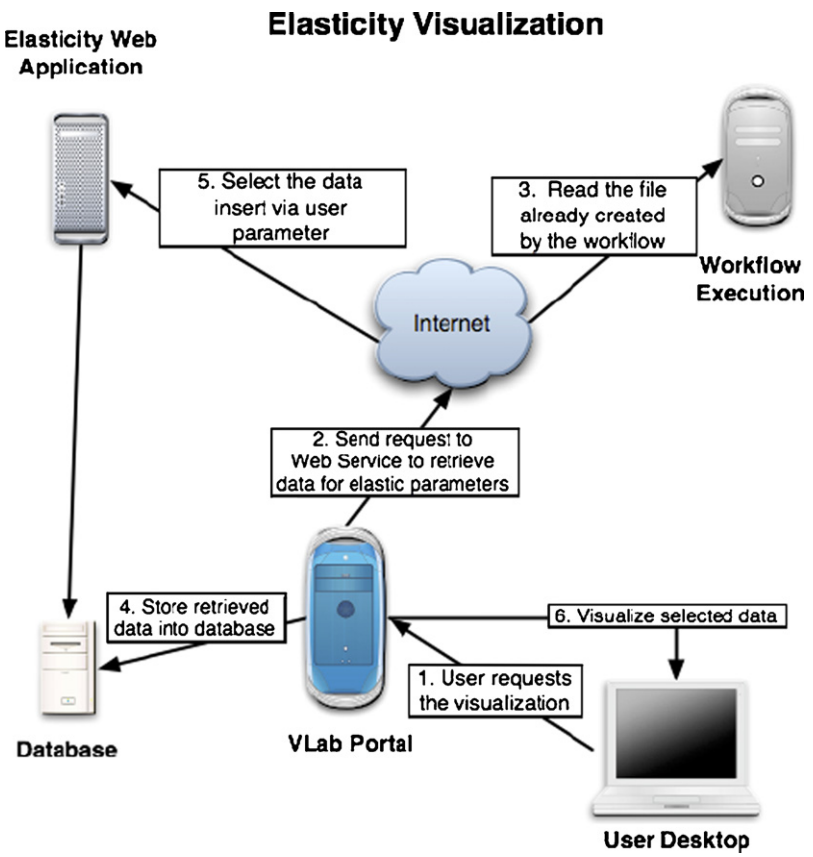

Fig. 1. An example workflow that occurs behind the scenes within the VLab portal. Users (far right) interact directly with the portal machine (bottom). The portal machine delegates requests for data, computation and visualization to other services.

1. Personalized views of the Grid ("my resources", "my services", "my information", and "my data") as well as tools to search for and discover data from community sources,
2. Mechanisms for conducting and archiving online computational experiments,
3. Natural interfaces for scientists to share their results with both their collaborators in a virtual organization and with the general public,
4. Simplified interfaces that allow new and inexperienced communities of users to have access to complex models and resources, and
5. Mechanisms for storing all details used to achieve a particular result in an unambiguous fashion, thus potentially revolutionizing the scientific peer review process.

We further note that these science gateways adapt very well to classroom and scientific training settings, providing educational tools for the next generation of scientists.

Web portals in general are distinguished by the following characteristics: they provide authentication, identity management, and access to personalized content. This distinguishes them from closely related but distinct web technologies such as Wikis and Content Management Systems. Science portals for running remote applications, managing remote files, and similar tasks, have been developed for a number of years. The early history (1996–2001) of the field is reviewed in Amin et al. (2006), which also serves as a convenient demarcation of an era, as component-based portals have revolutionized the field over the last 5 years. The Science Gateway at Global Grid (Wilkins-Diehr, 2006) and the Grid Computing Environments Workshops (Portals, 2006; Portals, 2005) at Supercomputing 2005 and 2006 give an excellent snapshot of the current state of the field.

Science portals have been developed for over a decade, and much progress has been made to standardize their architecture and component models. As can be seen in the above references, the so-called portlet component model, defined by the Java Specification Request

(JSR) 168 (Abdelnur et al., 2003), has been extensively adopted. Many open source JSR 168 containers have been implemented, with the GridSphere (Novotny et al., 2004) container serving as a very popular model in the scientific community. General purpose, pluggable Grid portlets for remote job submission, interactions with Grid information services, remote file management, and security credential management have been developed by the Open Grid Computing Environments collaboration (Alameda et al., 2006) and the Grid Portlets project (Russell et al., 2005). Most Java-based Grid portals (whether portlet-based or not) use the Java COG kit (Amin et al., 2004; Laszewski et al., 2001) to build their Grid clients.

Web technologies evolve rapidly. Portlet technologies, while providing an important component model for server-side, Java-based portals, must now adapt to many exciting new client-side technologies. We see clearly that development approaches such as AJAX (Asleson and Schutta, 2005) and related "Rich Internet Application" or "Web 2.0" technologies such as Direct Web Remoting (http://getahead.ltd.uk/-dwr), the Google Web Toolkit (http://code.google.com/webtoolkit/), the Yahoo! User Interface Library (http://developer.yahoo.com/yui/) and others are dramatically changing the level of interaction that users come to expect out of gateways. More generally, we see the emergence of "mash-ups", which allow different views of data to be easily overlaid on a display, as an important enabling technology. Mash-ups typically combine data and capabilities from a number of remote, third party services through well-defined programming interfaces (http://www.programmableweb.com/). By using event systems such as Adobe Flex (http://www.adobe.com/products/flex/) and social networking technologies like Ning (http://www.ning.com/), mash-ups may additionally be made both synchronously and asynchronously collaborative.

Although scientific mash-ups are in their infancy, they have the potential to significantly alter science portals. Not only do they integrate extremely well with the service-oriented architecture of Grids, but they also have the effect of democratizing the Gateway development process: well-designed programming interfaces and associated libraries to remote services enable busy scientists to develop their own powerful web interfaces.

3. State-of-the-art web interfaces

Web pages can be classified into two categories, static and dynamic, when describing their content. As indicated by its name, static content is encoded once and presents a constant, immutable display to the user. HTML is the quintessential example of static content. On the other hand, dynamic content is a function of external variables often encoded within the URL or HTTP Headers. For example, Java Server Pages (JSP) provides a scripting engine that can control the page HTML based on external variables. In the simplest case, the client-server model, dynamic content can be processed on either the client and/or on the server. Often, computational resources are lacking on the client, so that complex operations are left to the server. Nonetheless, an entire industry has evolved around client-side technologies; a prime example is JavaScript. One issue with client-side scripting is compatibility across browsers. It is hoped that "official" standards from the World Wide Web Consortium (W3C) and others will lead to software that has similar behavior across any browser that conforms to these standards. Of course, many browsers do not conform to the standards, leading to many complications. JavaScript still has many incompatibilities across the major browsers (Explorer, Fire fox, Safari), so one is limited to the most common operations, leaving browser-dependent operations (e.g., element positioning) to the control of libraries which hide the browser dependencies, or to server software. Most web sites and/or web applications employ a combination of client-side and server-side technologies to provide rich dynamic content to its viewers.

JSF is an acronym for Java Server Faces, an extension of the well-known and widely used Java Server Pages (JSP) (Bergsten, 2003). JSF provides an abstraction on top of JSP to allow rapid development of web interfaces. Fundamentally, JSF components are collections of reusable JSP codes, which are described by XML-like tags (Jacobi and Fallows, 2006).

3.1. Java server faces (JSF)

JSF is a dynamic web application framework that is similar to other dynamic web application templates like velocity (http://velocity.apache.org), Java Server Pages and Spring (http://www.springframework.-org). JSF provides dynamic content using Java servlet technology. Dynamic content is generated based on the request and response paradigm. Unlike static HTML pages, web application frameworks allow interaction with users. The server pages take requests, process them and respond to the user through a web interface. Besides its inherent virtues, we believe that JSF is very well suited for science gateway development, as we will discuss.

JSF applies the Model-View-Control (MVC) software design pattern (Singh et al., 2002) to decouple data models, action controllers and user interface widgets into

separate components. Within the framework of JSF, the model corresponds to a backing JavaBean: a piece of Java code that is responsible for managing the application's data. The backing bean itself is typically a client to a database or a remote web service. One of JSF's hallmarks is that these beans can be developed outside the JSF framework; that is, it is not necessary to import any JSF-specific code. This means that the beans can be reused in non-web based applications and can run (for testing purposes) outside of a Tomcat web server. This is accomplished through a design pattern known as "inversion of control". The controller corresponds to a JSF servlet that manages user requests. Finally, the view corresponds to web interfaces of JSF. This architecture separates the data access and user interaction. MVC encourages the reuse of backing beans within different applications.

The JSF framework provides a component model supported by user interface (UI) widgets in XML format to implement view pages. In contrast, JSP usually mixes and matches java code and HTML markups in the view pages. Instead, JSF completely separates view pages from beans by extending the HTML tags through a collection of JSF core tags such as <f:form/> and <h:commandButton/>. These tag instances bind attributes to backing beans defined in a configuration file (faces-config.xml) associated with the application. The configuration file also contains all the details of page navigation, making JSF a so called "model 2" framework (a variant of MVC), within which the logic of the model and the view are fully decoupled.

JSF portlets are built on top of the standard portlet API (JSR 168), enabling their deployment within various portal frameworks. Java portlet development generally requires the implementation of the portlet API within the web application. Fortunately, a tool referred to as the JSF portlet bridge simplifies the task. The JSF portlet bridge (http://portals.apache.org/bridges/) injects Java beans with the portlet API to eliminate the portlet programming phase by providing a servlet to deploy JSF applications as a portlet. It consumes the portlet API to simplify the deployment of any stand-alone JSF application as a portlet. The JSF bridge supports the MyFaces (http://myfaces.apache.org/) reference implementation of JSF, and may be incompatible with other reference implementations (e.g., from Sun Microsystems).

Portlet bridges have worked to optimize development time of the VLab portal. The PWSCF portlet uses JSF portlet technologies to start simulations in the remote servers. Using these abstraction utilities, it becomes straightforward to build simple interfaces and deploy them in portal containers like Gridsphere (http://www.gridsphere.org) and Jetspeed (http://portals.apache.org/jetspeed-2/), two of the most common containers in use for web interfaces.

One of the major advantages of JSF is to extend tag libraries with custom developed widgets. Our aim is to provide a set of Grid tags in JSF that provides Grid capabilities such as proxy credential, job submission, file operation, and workflow by means of multi-staged tasks. Grid tags are associated with Grid beans to access Grid services. Bean methods bind to tags with related property values.

3.1.1. Grid tag libraries

Building on these general JSF capabilities, our goal is to simplify the construction of science gateways out of reusable component parts. JSF technology helps to build user interfaces based on the component approach. User interface components, called UI Components, provide HMTL tags, web forms and tables. UI Component design is extensible for custom components. Standard JSF tags usually consist of visual components, although in our work, we also provide Grid tags as non-visual components.

Grid tags support single and multi-staged tasks. A multi-staged task is a Directed Acyclic Graph (DAG), with the associated tag name '*multitask*'. Consider the task of sending a file from glf1.ucs.indiana.edu to a machine on the Teragrid, cobalt.ncsa.teragrid.org, using the GridFTP 3rd party file transfer protocol. The required directory is created in a separate call to GridFTP. The file transfer is followed by a job submission. The files connected to standard in and standard out are specified within the script. Listing 1 shows how these tasks are specified using the JSF tags described in Nacar et al. (2006). Tags have also been provided to specify causal dependencies between tasks that are launched asynchronously.

```
<%@taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%@taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@taglib uri="http://www.ogce.org/gsf/task" prefix="o"%>
<f:view>
    <h:form id="vlabform">
    . . . . . .
<o:submit id="test" action="next_page" />
<o:multitask id="vlabtask" taskname="vlab" persistent="true">
<o:myproxy id="proxy" hostname="gf1.ucs.indiana.edu"
  port="7512"
    lifetime="2" username="manacar" password="******" />
<o:fileoperation id="taskA" command="mkdir"
    hostname="cobalt.ncsa.teragrid.org"
    path="/home/manacar/vlab/" />
<o:filetransfer id="taskB"

  from="gridftp://gf1.ucs.indiana.edu:2811/home/manacar/input"
```

```
   to="gridftp://cobalt.ncsa.teragrid.org:2811/home/manacar/vlab/
   input"/>
<o:jobsubmit id="taskC" hostname="cobalt.ncsa.teragrid.org"
      provider="GT2" executable="/home/manacar/script"
      stdin="vlab/input" stdout="vlab/result"
      stderr="vlab/error" />
<o:dependency id="dep1" task="taskB" dependsOn="taskA" />
<o:dependency id="dep2" task="taskC" dependsOn="taskB" />
</o:multitask>
</o:submit>
</h:form>
</f:view>
```

Listing 1: *Grid tag libraries are used to build a sample Web form. JSF will convert this into HTML. The first two* <%@taglib%> *tags import standard JSF tags* (*such as* <f:view/>). *The third set of imported tag libraries (corresponding to all tags with the "o"prefix, such as* <o:submit/>) *are our Grid tags*.

The Grid tags found in Listing 1 match up with corresponding Grid beans that implement Java Cog Kit abstractions (Amin et al., 2006), which wrap Grid services to support an additional layer of client programming. Grid tags and beans are described in detail at (Nacar et al., 2006). Grid tags simplify Grid portlet development and hide the complexity of the Grid clients from the developer. They also handle user sessions to monitor and track user requests. Each request is given a unique identifier that combine timestamps in the particular session.

Grid tags allow users to retrieve metadata about submitted jobs, such as job status, completion time, output file location, etc. Users can also keep track of running jobs, and they can archive metadata related to completed jobs, output files and other relevant data. This system also provides maintenance features on the archived data like removing, modifying metadata or resubmitting the old jobs.

4. Workflows

Perhaps the single most important feature of VLab is its project execution management through workflows. Workflows describe the complex web of interconnected services and tasks, taking into account dependencies, which must be completed for even the simplest problems. In its simplest form, a workflow is a collection of interconnected tasks (simulation, storage, retrieval, analysis, visualization): the output of one task feeds the input of other tasks. These tasks are selected through access to a portal. As explained in (Silva et al., 2007) VLab handles six basic types of workflows: (1) simple execution of PWscf codes (http://www.pwscf.org), (2) pressure point sampling to determine a static equation of state (EOS), (3) EOS at finite temperature, (4) computation of elastic constants (Cij), (5) computation of Cij at finite temperature, and (6) post-processing and visualization. Each of these workflows can be described in terms of smaller workflows, and so on recursively. The individual components of these workflows are implemented as Web Services under the control of the so-called Project Executor. A Task Dispatcher is responsible for selecting the appropriate computational resources (that are distributed) and initiating the desired task. One of the features of the system, is the ability to execute tasks concurrently, a necessary requirement given that anywhere from a few to thousands of runs are executed to perform some types of calculations (e.g., phonon computations for high temperature, Cij). Long running tasks are tracked via a receipt system. A task that runs longer than the duration of an unattended web transaction will request a receipt (i.e., metadata) from the running service. This receipt enables the originator to query the task at a later time. Since any task can launch subtasks, the receipt structure is recursive. It contains all necessary metadata to reconnect with the particular server executing the task. More details on this process can be found at (Silva et al., 2007). Figs. 1 and 2 illustrate two visualization-related workflows controlled by the portal. In both cases, it is assumed that the relevant computation has been performed, under the control of one of the PWscf execution workflows. As seen from the figures, each workflow has multiple steps, and in both cases, the user retrieves an image generated on a remote server. The elasticity visualization accesses data from previously executed elasticity calculations, which is stored in a database by workflow (4) above.

5. Automatic web service generation

The ever increasing size and complexity of scientific data has forced developers to turn to more powerful alternatives for visualizing results of computation and experiments. Changing trends in technology have also led to a high demand for service-oriented applications such as the portals and workflows described above. Unfortunately, developing the necessary workflows of web services is a meticulous, challenging and repetitive task. While services vary in complexity and quality they are all constructed from some code, responsible for the actual task to be executed on behalf of a requestor, and wrapped up in SOAP messaging protocols. Each time a new service is developed, mistakes are inevitably made, sometimes multiple times, due to the many details writing a web service entails. Generating web services automatically is one of the goals of the VLab project.

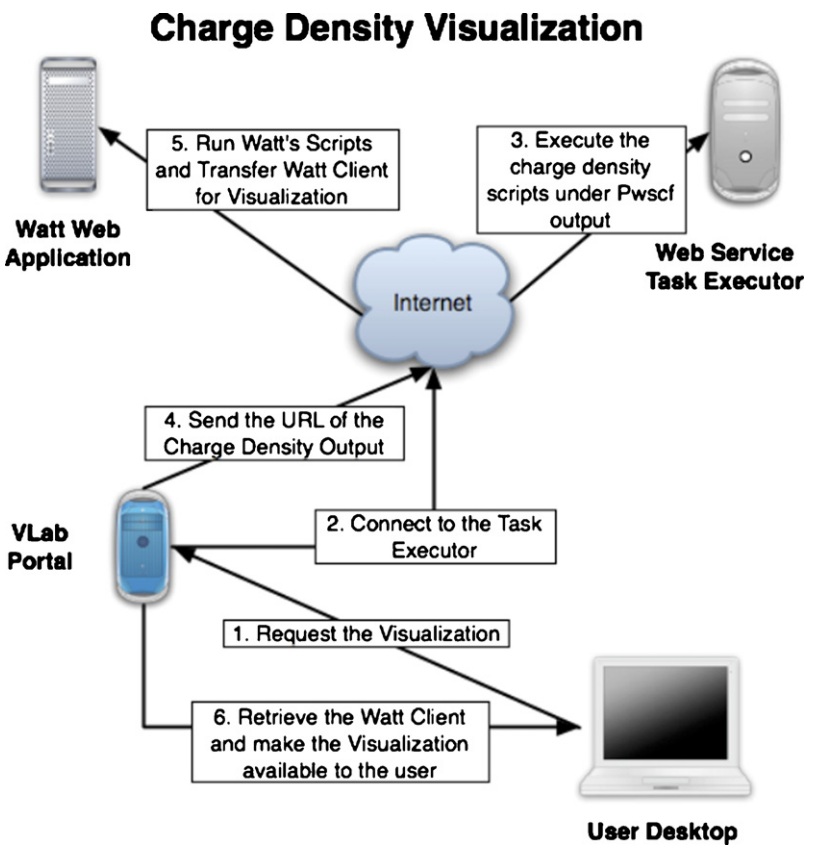

Fig. 2. The charge density visualization workflow. The WATT Web Application (top left) is a C++ application utilizing VTK libraries. The visualization application is first written in Tcl script and compiled into C++ using WATT into C++.

Such a capability would empower the researchers to not only to work within the confines of existing services and workflows, but also to construct them dynamically under changing conditions.

In the last decade, advances in scripting languages and visualization libraries have significantly decreased the development time of new applications. Scripting languages allow developers to rapidly prototype software by hiding the fine implementation details from the programmer. The use of dynamic typing and automatic memory management (garbage collection) significantly speeds up development time at the cost of execution time. Many scientific libraries are published with scriptable interfaces, allowing scientific software developers to focus less on the fine details of programming and more on the scientific content of their applications.

We present two approaches to automate the generation of web services that address the aforementioned issues. The first approach uses Ruby to read in simple description files and translate these into Java Web Services. A second type of web service generator is concerned with the translation of Tcl scripts (that serve as a front end to several visualization packages, notably the Visualization Toolkit (VTK) and Amira, into C++ visualization services. The Web Automation and Translation Toolkit (WATT) converts a Tcl script written to drive VTK into a standards-compliant web service. For both approaches we describe the conversion process and an example of how they are used within the VLab portal.

5.1. Web service generation with ruby

Software development involved writing JSF pages, java beans, web services, and workflows. All these tasks were error prone, wasting much of the developer's time debugging the programs produced; yet many of the programs had elements in common: structure, variable names, method and class names. Using one code from template allowed new code to be quickly generated via cut and paste, with quick changes. This process leads to errors. While variable names are well encapsulated within well-written Java classes, the same cannot be said for method names. Changing a method names necessitates changes to potentially many program

files. Duplication of any information within a code or framework complicates the task of rearranging code, changing the structure of a database, or writing error-free code. Several web services interact with each other via workflows. Generation of these workflows requires the creation of scripts using an extended version of Javascript. Although this is easy to do, mistakes are easily made, so automation of this process is very useful. Scripts occur at all stages of the VLab project. They are used to control ant (http://ant.apache.org), maven (http://maven.apache.org), hpsearch (http://hpsearch.org) or shell scripts. For example, if a file is named *browser*, the label *browser* might appear in several supporting scripts and JavaBeans, or JSP/JSF pages and their configuration files. If the word *browser* would to be changed to *web-browser* to increase clarity, it follows that this change would have to propagate to all files that use it. The natural tendency is thus to avoid such changes, leading to software eventually becomes quite unreadable and unmanageable.

Ruby as an interpreted language is not the fastest; however, it is probably one of the easiest to learn and is extremely flexible. We have therefore chosen Ruby as the means to generate the web services and other code necessary to the VLab framework. Ruby has the ability to read in files in so-called *yaml* format. This format is similar to that of Java Property files. A key difference however, is that property files only contain a single hash table, while *yaml* files can hold arbitrary combinations of array lists and hash tables, and they can be arbitrarily nested within one another. It thus becomes possible to describe Java Beans, Web Services, HPSearch scripts, etc. in a very compact way, avoiding duplication.

As a simple example of the use *of yaml* files, we describe the automatic generation of Java beans. It is important to realize, that we do not seek to implement all the features of java beans; only the features that are used most of the time within our components. The advantage of this approach is the degree of compactness. Many methods in java beans are simple set and get methods, and these can be generated automatically. Specific methods are still written by the user. They are contained in files with a *.tpl*" extension. These are included within the bean class.

Consider the following *yaml* input file:

```
class : “FileBean”
types :
String :
– toDir
– fromDir=“/path/to/dir”
int : [x=3, y=2, z=4]
Context : [“ctx”]
package : ogce.gcf.context
import :
– java.util.*
– java.lang.*
– ogce.gcf.context.*
templates : [common.java.tpl]
```

Listing 2: *Example input file in yaml format with enough information to build a java bean*.

A ruby program reads this input file and creates a java bean with the following characteristics: its name and class are *FileBean* (the duplication between the file name and the class name has been eliminated). The package is ogce.gcf.context, two packages are imported, there is one template, and there are five variables for which read and write accessors are created. Three of these variables have default values. If additional methods, change of arguments, etc. need to be implemented, it is only necessary to modify the script and rerun the underlying ruby program.

The generation of web services is more complex. The generator creates the client, the server, a list of scripts used for compilation, linking, inclusion of the service within the Tomcat framework and more. In addition, client and server code are each output to their own sub-directory structures. Rather than describe in detail the output of the web service generator, we provide the input file to give the reader an idea how all the elements required of a web server are listed with no redundancy.

Once again, we specify a single base name for the service with related names for directories, packages, client and server code, etc. An example of a *yaml* file input for web service generation is given below:

```
tomcat : /home/u6/users/erlebach/src/apache-tomcat-5.5.12
name : script
namespace : ws
port : “8090”
hostname : gf8.ucs.indiana.edu
methods :
–
name: executeHash
arg: [“String”]
return: String
–
name: executeString
# script, args, output, timestamp
arg: [“String”, “String”, “String”, “String”]
return: String
```

Listing 3: *Example input file in yaml format with the necessary information to write a webservice, including necessary scripts to bind the resulting webservice to the Tomcat container. The user is responsible for writing the client and server core routines*.

The top section describes parameters related to the Tomcat container, while the methods section describes the web service methods accessible by users and remote

machines (executeHash, executeString). The name of the web service is script. The script generator creates a directory for the web service, one for the client, and one for a collection of scripts necessary for compilation, execution, and control of the Tomcat container. Java code is created for the server and for the client. The user is responsible for the implementation of the server and client code. However, all wrappers and SOAP-dependent stubs are generated automatically. The code to be modified contains *no* SOAP-related elements. The Java class for the service is Script (note the capital S), the namespace is script, the implementation class is ScriptImpl, and so on. The supporting scripts for compilation, execution, etc. are constructed on the basis of templates (shell scripts) in which necessary keywords are defined as:

```
service_class=SERVICE_CLASS
client_class=CLIENT_CLASS
top_dir=WEB_SERVICE
url=URL
```

where the variable names on the right-hand side are defined within the web service generator. All the scripts contained in a templates/directory are processed and the template variable names replaced by names as specified in the generator. These names most often contain the base web service name, in this case, script.

5.2. Automatic generation of visualization web services

At this time there are few visualization web services, mostly because within this class, web services are quite inefficient and difficult to develop. However, remote rendering tasks are ideally suited for web services when interactivity is not a key concern. We are interested in providing users with the maximum amount of flexibility regarding their visualization tasks without constraining them into design choices imposed by the designers of VLab. It is necessary to allow users to create their own visualization scripts and provide tools to automatically transform these scripts into web services. Among the various visualization packages that are available, we have chosen to work with the Visualization Tool Kit (VTK), a free and very powerful visualization package. Its APIs are written in C++, and are accessed from within C++ programs. The authors of VTK have made available bindings of the VTK library to the Tcl scripting language (http://www.tcl.tk). Tcl is a simple interpreted language used as a front end by several other software frameworks including Amira (http://www.amiravis.com), VMD (http://www.ks.uiuc.edu-/Research/vmd/) and Graph Viz (http://www.graphviz.org/). The objective is to develop a tool to translate Tcl scripts into full-fledged web services without user intervention.

We present the Web Automation and Translation Toolkit (WATT), which takes as input a Tcl script and generates a C++ program with appropriate SOAP classes (also in C++) to form a functional web service (see Fig. 3). The SOAP classes are compiled by the gSOAP compiler (http://www.cs.fsu.edu/∼engelen/-SOAP.html) and combined with the translated program into a binary executable service (Bollig et al., 2006; Jensen et al., 2006).

5.2.1. WATT overview

The Web Automation and Translation Toolkit (WATT) contains a compiler and associated tools to automate the generation of a web service from a stand-alone desktop application. The compiler uses information from the header files of a set of C++ libraries to convert a Tcl script into a C++ program, in effect creating a scripting layer on top of the library. The compiler also contains a type inferring mechanism to assign type information to values in the script based on their context and use before runtime. In addition to the core C++ program translated from Tcl, WATT has the ability to generate a configuration file and header files specifically for the gSOAP compiler (Engelen, 2003), whose output can then be compiled with WATT's output to generate a webservice. At this time, WATT does not support the full Tcl language; we have only implemented keywords essential for our service development. However, functionality does exist which allows us to perform generic floating-point arithmetic, manipulate complex data types (i.e. objects) and evaluate method/procedure calls. As we continue to use WATT in the expansion of the VLab portal, additional support will be added.

The WATT compiler (Jensen et al., 2006) is written in Objective CAML (OCaml) (http://caml.inria.fr/ocaml/). OCaml is a strictly typed functional programming language that guarantees type safe programs and allows object-oriented design. With compilers that reduce it to machine level byte-code, typical performance of OCaml is on the level of C++, but without the frequent memory access errors (i.e. segmentation faults) and other type-related problems suffered by its counterpart. Many features of OCaml made it a prime candidate for the WATT compiler, including internal constructs like the global Hash-Table ("Hashtbl"), built-in exception handlers and type matching utilities. As WATT compiles Tcl, it uses matching utilities to determine type information for tokens, statements, etc. Once discovered, WATT stores the paired type and token information within the Hashtbl, or alternatively raises an exception if type can-

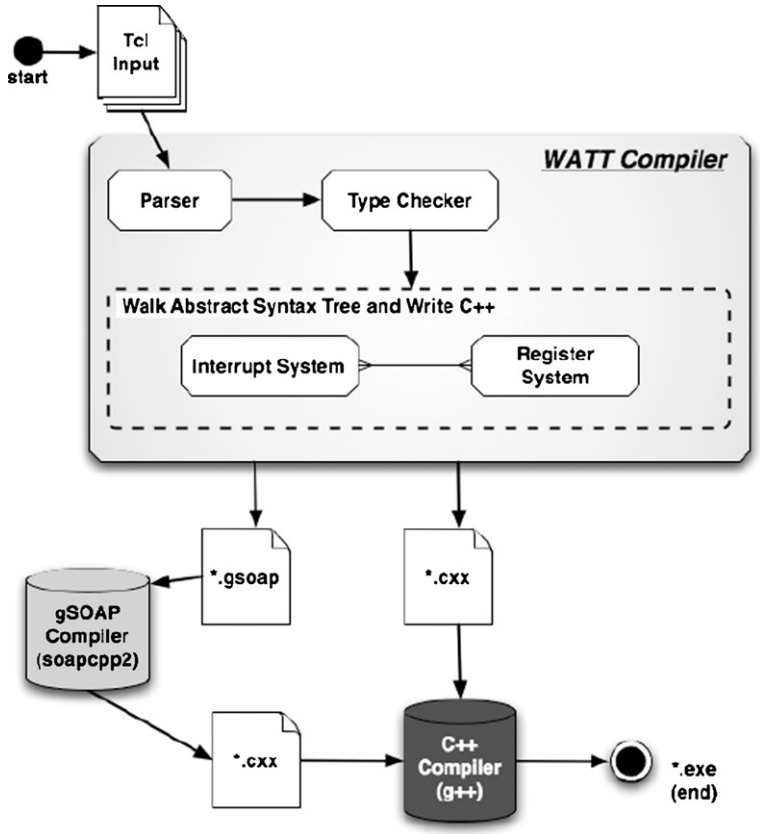

Fig. 3. Flow of information through the WATT system.

not be determined. Since OCaml allows us to manually override exception messages, the WATT compiler piggybacks compilation errors on top of built-in exception handling to provide verbose messages regarding unrecognized classes, incorrect Tcl syntax, etc.

WATT initiates a typical conversion from a Tcl script by running an integrated parser and type-checker. The object-oriented extensions to the Tcl language used by most software packages create ambiguities between regular Tcl commands and instantiation and method invocation statements, so the type of the token must be used along with the context of the code to determine how the object in question is being used. The type information for each token is preserved in the abstract syntax tree for use when the code is re-written in the output language.

Once the type of each statement (command, instantiation, or invocation) has been correctly assessed, WATT again walks the abstract syntax tree to determine the type of variables not explicitly declared (such as arguments to procedures) and deduces the correct signatures for user-defined functions. The return type of each function is determined from the last expression in the function definition. However the types of the function arguments can also be ambiguous within the function body. In these special cases, WATT delays typing the function arguments until later in the script once the function has been called. The types of the values passed during the call are then used to determine type. The entire typing process finishes as the program is re-written in the output language when it must evaluate complex statements and return types to validate correct usage and typecasts in assignments.

Classes are described to WATT as instances of a Register class. Each instance stores information regarding keyword or class name, methods, constructor and necessary headers to include in C++. As the compiler encounters statements using these registered keywords, it evaluates the statement with type information available in the registry. In a way, registers allow us to provide WATT with direct translations from Tcl to C++. For example, consider the evaluation of the Tcl statement

```
myObject methodCall $arg1 $arg2
```

translated into C++. It becomes

```
myObject→methodCall (arg1, arg2)
```

If one assumes that myObject is of type myObjectType and can be included from myObjectType.h, we generate the following register for WATT to correctly perform translation:

```
register_class "myObjectType" {
    class_methods = [{method_name="methodCall"
        method_return = Void;
        method_args = [Int; Float]; } ];
    class_constructor = {
        method_name = "new";
        method_return = Instance "myObjectType";
        method args = []
    };
    class_include = "MyObjectType.h"
}
```

Listing 4: *Illustration of Objective CAMEL code that implements a register* (*essentially an object class*).

From this register, WATT can determine that methodically is a valid method call and that arg1 is an integer and arg2 is a float. The task for WATT is to complete the translation of class method calls by adding the standard arrow ("→") and parentheses to the statement for correct C++ syntax.

Unfortunately, in many cases keywords cannot be directly translated. For example, "add" – as in "add $x 2" – can be directly translated to the "+" operator in C++, but the ordering of arguments must be switched to read "x + 2". To handle this case, we allow an interrupt function to be registered with WATT as follows:

```
let test = function x → match x with Command ("add", _) → true
  and redo = function x → match x with Command (name, args) →
    let s = Buffer.create 0 in
      begin
          append s (arg_to_string (List.nth args 0));
          append s "+";
          append s (arg_to_string (List.nth args 1));
          Buffer.contents s
      end
    in
      ignore (register_interrupt Command_to_string {
          interrupt_test=test;
          interrupt_func=redo
      } );
```

Listing 5: *Objective CAME code that implements an interrupt routine*.

The interrupt system is called before the standard registers and uses matching to determine the association, if any, between statements and interrupts. When the compiler matches a statement to the arguments of the interrupt function, it calls the function with the statement tokens as arguments. The interrupt is then in charge of rearranging the statement or injecting new information into the resulting string buffer. A string buffer is returned for all interrupts and is sent directly to the output file, bypassing the standard registers.

5.2.2. *Visualization web services*

The creation of web services with WATT is somewhat similar in concept to the separation of content and presentation when creating HTML pages using CSS. Content creation and layout can proceed independently, according to set standards, which guarantees a clean page for the end user. HTML content creation corresponds to core computational or visualization components responsible for reading, rendering, and presenting scientific data. CSS layout corresponds to the service component that is responsible for encapsulating data on the server, marshalling it across a network, and unpacking it on the client. Together, the user is presented with working web services for a wide variety of visualization tasks.

Provided a consistent interface is maintained, the two components of service-oriented applications can be implemented independently. Scientists concentrate on creating Tcl code that executes on the client machine while service-oriented developers work on wrapper code to provide security, throughput and flexibility. Automatic generation of the services allows standard interfaces to be developed that guarantee inter-application functionality and furthermore enables the generic reuse of software across multiple workflows. Providing a standardized method of communication between service providers and clients can also streamline the integration of middleware technologies, since modifications need only be made to the interface of the service, not to the core body of the application code.

Although WATT was originally designed to compile standard Tcl into C++ services, the compiler also optionally handles Tcl scripts that access the VTK libraries. When generating VTK services, WATT assumes that a number of web methods are available for all visualization services, for example, methods to export bitmap images of the rendering window and transformation methods. Thus the service component of the application has been completely abstracted from the visualization component. No knowledge of the service implementation is needed to create a web-based visualization tool (Bollig et al., 2006).

To connect to the visualization services created through WATT, we created a generic client to display the resulting bitmaps and simulate interaction with the image. The WATT client is a Java WebStart application that both displays a bitmap and draws a 3D bounding box around the atoms. This box, which overlays a background bitmap image, can be interactively manipulated by the user with the use of a mouse. Once the mouse is released, the box orientation parameters are sent to the remote service, which recomputes the image with the alignment, returns its bitmap for the client to display. The

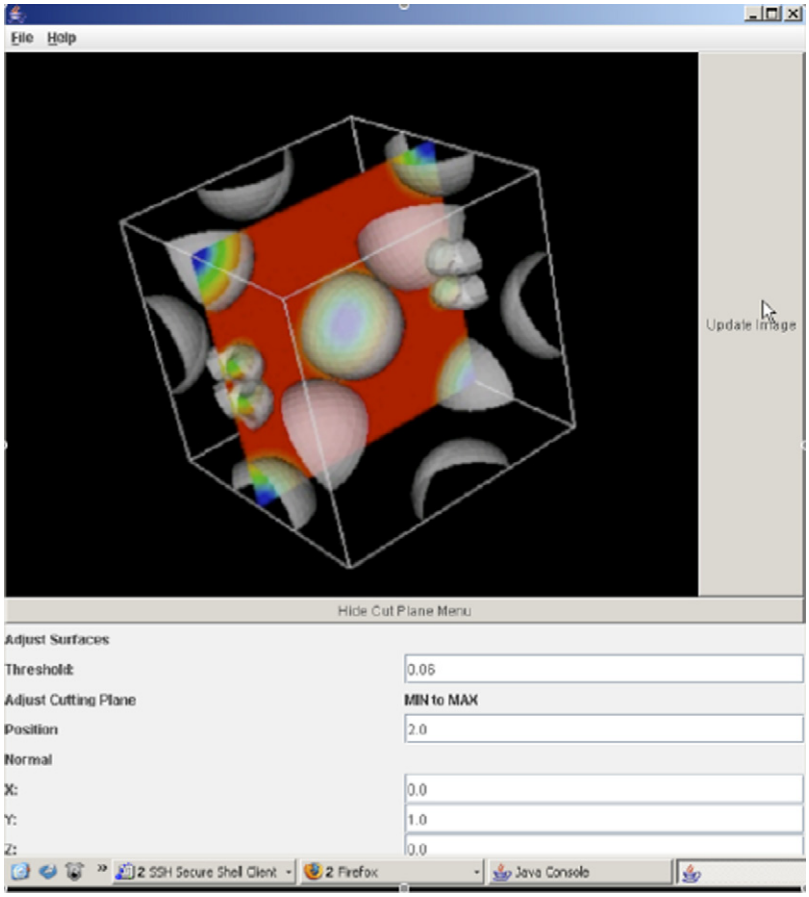

Fig. 4. Visualization Web Service to visualize charge density plots obtained from PWscf. The service was created with the help of WATT. The figure shows the charge density in an eight-atom cell of $Mg_{1-x}Fe_xO$, with $x = 0.25$.

WATT Client also provides facilities for image scaling and translation. Because the WATT Client runs on any machine with Java WebStart, it is a platform-independent viewer. In its current state, the WATT Client is useful for testing the ability of WATT services to generate images and accept remote requests.

5.2.3. Example: charge density service

We illustrate the use of WATT by processing a set of data produced by a phonon calculation: the map of the charge density around a set of atoms. The resulting data file from a single often consumes over 200 MB of storage, so that file transfer to a local machine is not the most desirable option. Instead, users first created a Tcl script to load the data, add an isosurface, and slice the data with a cutting plane. The Tcl script was compiled with an early version of WATT (and subsequently the gSOAP compiler) and deployed the service on a visualization machine at the Minnesota Supercomputer Institute. VLab users simply clicked a button in the portal to download and run the WATT Client, and were presented with a 3D, interactive application with controls for rotation, translation, and cutting plane (Fig. 4).

WATT and the WATT Client are particularly suited to the needs VLab' for the following reasons:

- WATT provides remote visualization services, essential when working with large datasets.
- The web service-based visualization model conforms to existing service-oriented architectures. (The conversion and transfer of data to the visualization server are handled by other web services.)
- In principle, WATT allows the visualization service to be developed quickly. The original version of the charge density code was written within 100 lines of Tcl code.

6. Conclusion

We have provided in this paper an overview of recent developments of the VLab, the Virtual Labora-

tory for Planetary Science and Materials. The work has progressed along several fronts. A portal, based on GridSphere, is used as a front end for the VLab users. Through a series of portlets, users execute prewritten workflows that control execution of a series of calculations in a carefully orchestrated manner. These workflows take care of computation, analysis and visualization. Through our work, we have recognized the need to provide users with the capabilities to change their environment and construct their own tools, while understanding that, for the most part, their interest is in the science and not in the low level details. They are not interested in standards such as JSF, AJAX, Web 2.0, SOAP, and Web Services. Rather, they wish to run codes, analyze their results and visualize their data. Previous experience suggests that whatever tools the users are provided, they will always ask for more features, different data paths, additional data views, different workflows, and more. We have therefore started to develop tools that construct the necessary components used to build up VLab, including Java Beans, Web Services, and Workflows. The idea is to enable users to write out tasks in a simple language, and let the tools take care of transforming these requests into compiled code ready to function in a distributed environment. The Ruby web service generator and WATT, the visualization web service generator are first steps in this direction. Users must be able to construct their own workflows. A first approach is the use of JSF pages that provide users with the means to construct simple workflows with dependencies from within a web browser.

In the near future, we will continue to work on the WATT compiler and apply it to a range of practical problems of increasing complexity. The real challenge will be to develop a selection of client software to interface with a broad range of visualization services. Is it possible to create a universal client, or will specialized clients, perhaps generated automatically based on the server specifications be the best approach? In the near future, we will concentrate on visualizing the data produced within the VLab framework. On project will involve a graphical interface to the metadata generated during the execution of VLab workflows, allowing users to track details of past projects and projects still executing. This will provide the means to track the various activities of VLab across a potentially large distributed network. A second project will be concerned with accessing the potentially hundreds of data files produced by phonon calculations and presenting this information in a meaningful way. The current approach will explore the means to allow users to control interactively which variables to plot, how many plots, along with additional controls. Whether this will be done using AJAX within the web browser or Java remains to be seen.

Acknowledgments

This work is supported by the National Science Foundation's Information Technology Research (NSF grant ITR-0428774, 0427264, 0426867 VLab) program.

References

Abdelnur, A., Chien, E., Hepper, S. (Eds.), 2003. Portlet Specification 1.0. http://www.jcp.org/en/-jsr/detail?id=168.

Adobe Flex 2 Web Application Development Software. http://www.adobe.com/products/flex/.

Alameda, J. et al., 2006. The Open Grid Computing Environments collaboration: portlets and services for science gateways.

Amin, K., Hategan, M., von Laszewski, G., Zaluzecw, N.J., 2004. Abstracting the Grid, 12 Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2004), pp. 250-257.

Amin, K., von Laszewski, G., Ali, R.A., Rana, O., Walker, D., 2006. An abstraction model for a grid execution framework. J. Syst. Architect. 52 (2), 73–87.

Asleson, R., Schutta, N.T., 2005. Foundations of Ajax. Apress, p. 296.

Bergsten, H., 2003. JavaServer Pages. O'Reilly Media, p. 664.

Berman, F., Fox, G., Hey, T. (Eds.), 2003. Grid Computing: Making the Global Infrastructure a Reality. John Wiley & Sons, Chichester, England.

Bollig, E.F., Jensen, P.A., Erlebacher, G., Yuen, D.A., Momsen, A.R., 2006. A Responsive Client for Distributed Visualization, Eos Trans. AGU. Fall Meet. Suppl., Absract IN14A-07.

Bukowinski, M.S.T., 1976. On the effect of pressure on the physics and chemistry of potassium. Geophys. Res. Lett. 3, 491–494.

Cohen, R.E.E., High performance computing requirements for the computational earth sciences.

Engelen, R.V., 2003. gSOAP and Web Services. C/C++ Users Journal 23 (2), 67–13.

Foster, I., Kesselman, C. (Eds.), 1999. The Grid. Blueprint for a New Computing Infrastructure. Morgan Kaufmann Publishers, Inc..

Foster, I., Kesselman, C. (Eds.), 2004. The Grid 2: Blueprint for a New Computing Infrastructure. Morgan Kaufmann.

Foster, I., Kesselman, Tuecke, S., 2001. The anatomy of the grid: enabling scalable virtual organizations. Intl. J. Supercomput. Appl., 200–222.

GCE05, Grid Portals Workshop, 2005. Seattle, Washington.

GCE, 2006. http://www.cogkit.org/GCE06/GCE06/Program.html.

Google Web Toolkit. http://code.google.com/webtoolkit/.

Jacobi, J., Fallows, J., 2006. Pro JSF and Ajax: Building Rich Internet Components. Apress, p. 464.

Jensen, P.A., Bollig, E.F., Yuen, D.A., Erlebacher, G., Momsen, A.R., 2006. Automatic Generation of Remote Visualization Tools with WATT, Eos Trans. AGU. Fall Meet. Suppl., Absract IN14A-06.

Jordan, T.H., Maechling, P., 2003. SCE/CME collaboration The SCEC community modeling environment: an information infrastructure for system-level earthquake science. Seismol. Res. Lett. 74 (3), 324–328.

Laszewski, G.v., Foster, I., Gawor, J., Lane, P., 2001. A Java commodity grid kit concurrency and computation. Pract. Exp. 13 (89), 643–662.

Nacar, M., Pierce, M., Erlebacher, G., Fox, G., 2006. Designing Grid Tag Libraries and Grid Beans Second International Workshop on Grid Computing Environments, GCE06 at SC06, Tampa, FL.

Novotny, J., Russell, M., Wehrens, O., 2004. GridSphere: a portal framework for building collaborations. Concurr.—Pract. Exp. 16 (5), 503–513.

Russell, M., Novotny, J., Wehrens, O., 2005. The Grid Portlets Web Application: A Grid Portal Framework, Parallel Processing and Applied Mathematics.

Science Gateways Workshop at Global Grid Forum 14. http://www.gridforum.org/GGF14/-qqf events schedule Gateways.htm.

Silva, C.R.S.d., Silveira, P.R.C.d., Karki, B., Wentzcovitch, R.M., 2007. Virtual Laboratory for Planetary Materials: System Service Architecture Overview Phys. Earth Planet. Int., companion paper, this volume.

Singh, I., Stearns, B., Johnson, M., 2002. Designing Enterprise Applications with the J2EE Platform. Addison-Wesley.

Wilkins-Diehr, N., 2006. Science Gateways—Common Community Interfaces to Grid Resources.

Yahoo! User Interface Library Web Site. http://developer.yahoo.com/yui/.