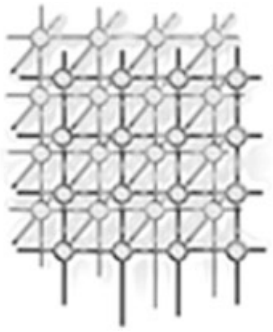
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2007; **19**:1717–1728
Published online 7 June 2007 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1199



VLab: collaborative Grid services and portals to support computational material science

Mehmet A. Nacar[1], Mehmet S. Aktas[1], Marlon Pierce[1,*,†], Zhenyu Lu[2], Gordon Erlebacher[2], Dan Kigelman[3], Evan F. Bollig[3], Cesar R. S. da Silva[3], Benny Sowell[3] and David A. Yuen[3]

[1]*Community Grids Laboratory, Indiana University, 501 N. Morton, Bloomington, IN 47404, U.S.A.*
[2]*School of Computational Science and Information Technology, Florida State University, 400 Dirac Science Library, Tallahassee, FL 32306-4120, U.S.A.*
[3]*Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, MN 55455, U.S.A.*

SUMMARY

We present the initial architecture and implementation of VLab, a Grid and Web-Service-based system for enabling distributed and collaborative computational chemistry and material science applications for the study of planetary materials. The requirements of VLab include job preparation and submission, job monitoring, data storage and analysis, and distributed collaboration. These components are divided into client entry (input file creation, visualization of data, task requests) and back-end services (storage, analysis, computation). Clients and services communicate through NaradaBrokering, a publish/subscribe Grid middleware system that identifies specific hardware information with topics rather than IP addresses. We describe three aspects of VLab in this paper: (1) managing user interfaces and input data with JavaBeans and Java Server Faces; (2) integrating Java Server Faces with the Java CoG Kit; and (3) designing a middleware framework that supports collaboration. To prototype our collaboration and visualization infrastructure, we have developed a service that transforms a scalar data set into its wavelet representation. General adaptors are placed between the endpoints and NaradaBrokering, which serve to isolate the clients/services from the middleware. This permits client and service development independently of potential changes to the middleware. Copyright © 2007 John Wiley & Sons, Ltd.

Received 30 December 2005; Revised 23 August 2006; Accepted 13 February 2007

KEY WORDS: cyber infrastructure; portal; science gateway; computational material science; messaging middleware; collaboration

*Correspondence to: Marlon Pierce, Community Grids Laboratory, Indiana University, 501 N. Morton, Bloomington, IN 47404, U.S.A.
†E-mail: mpierce@cs.indiana.edu

Contract/grant sponsor: National Science Foundation; contract/grant numbers: ITR-0428774, 0427264, 0426867 and 0330613

1. INTRODUCTION: GRID ENABLING MATERIAL SCIENCE APPLICATIONS

The Virtual Laboratory for Earth and Planetary Materials (VLab) is an interdisciplinary research collaboration, funded by the National Science Foundation, whose primary objective is to investigate planetary materials at extreme conditions based on *ab initio* computational techniques to better understand the processes that create Earth-like and other planetary objects. Such calculations typically involve hundreds or thousands of computer runs. These runs occur in stages, with complex interactions among them, and are often managed by several researchers. To address challenges in collaborative and distributed computing, the VLab team consists of researchers in computational material science, geophysics, scientific visualization, Grid computing, and information technology. Additional information on VLab is available from the VLab Web site [1].

Some of the many problems that VLab must address include the ability to create input files through portals, submit jobs, store and retrieve the job input and output data on demand, analyze and visualize the data, and store the data. These tasks must be possible in a distributed environment and the flow of information must be accessible to multiple collaborating researchers, distributed geographically. An additional constraint on our system is that it must be robust, i.e. fault tolerant. When working in a complex multi-user environment, it is inevitable that some components will fail. However, these failures should not affect the work of the individual researcher. Thus, we have chosen to connect the users of the systems (referred to as clients) and the various tasks (storage, visualization, analysis, job submission, etc.) requested by the users (wrapped into Web Services) using NaradaBrokering [2], a middleware with many of the required features built-in.

In its initial phase, VLab follows a well-established pattern for building Grids: application codes on remote machines are accessed securely through Grid services through a browser portal. This follows the common three-tiered architecture [3]. A user interacts with a portal server through a Web browser. The portal server in turn connects to remote Grid services that manage resources on a back-end system that provides the computing power for running the codes. However, the longer term research goal is to go beyond these traditional approaches. The distinguishing feature of our research is the use of the publish/subscribe paradigm, which completely decouples the clients from the services. Users have no knowledge of the resources allocated to their requests, although they maintain the ability to monitor task progress.

Many of VLab's workhorse simulation codes are included in the 'Quantum Espresso' package developed by the Democritos group [4]. As a starting point towards developing an automated Web Service workflow, we consider PWscf, the Plane-Wave Self-Consistent Field [5] code within the Espresso suite. PWscf is a parallelized application, often submitted to supercomputing resources via a batch queuing system. Common Grid technologies such as WS-GRAM, Reliable File Transfer, GridFTP, and Grid security, from the Globus Toolkit [6], provide the means to interface external applications with several popular schedulers, such as LSF, Condor, and PBS. Several additional problems must be addressed and are discussed in this paper: (a) managing user inputs as persistent, archived, hierarchical project metadata (Section 2); (b) simplifying and monitoring complicated, multi-staged job submissions using Grid portal technology (Section 3); and (c) integrating VLab applications with Grid messaging infrastructure [2] to virtualize resource usage, provide fault tolerance, and enable collaboration (Section 4).

DOI: 10.1002/cpe

2. MANAGING USER INTERFACES AND INPUT DATA

Job submission tasks are broken down into the following components: (1) provide a user front-end to facilitate the generation of a PWscf input file; (2) move the input file to a back-end resource, usually the computer that will run PWscf; and (3) run PWscf. Grid Web portals [3] are used to manage the collection of processes that define the submission task. This is a classic Grid portal problem. As detailed in a follow-up issue to [3] (currently in preparation), most of the existing Java-based portals are developed around the 'portlet' approach [7]. Portlets provide a consistent framework by which Web applications may be packaged and deployed into standard-compliant portlet containers. A portlet is typically a single, mostly self-contained application, such as the set of Web forms, Java code, and third-party jars needed to submit the PWscf code. Portlets are deployed into a portlet container, which is responsible for general purpose tasks, such as handling user login, providing a layout manager that arranges the portlets on the user's display, determining the user's rights to access particular portlets, and remembering the user's customizations (such as page arrangements and skin colors). We follow this approach and adopt the GridSphere [8] portlet container for VLab deployment. Our choice of standard-compliant portlets leaves open the possibility of changing containers in the future (such as uPortal or Jetspeed2), and allows other collaborators who may prefer these different containers to use the VLab portlets.

Portlets may be developed using several different Java Web technologies. For VLab, we decided to test portlet development with Java Server Faces (JSF) [9]. JSF is a model–view–controller framework for building Web applications that provides three very important advantages for all applications. First, Web developers do not need to explicitly manage HTTP request parameters. This eliminates the dependency of the backing Java code on specific name attributes in the HTML <input> tags. Second, the Web form's business logic is encapsulated in simple JavaBeans. Each HTML <input> parameter in a Web form is associated with a property field in the JavaBean that manages the page. Finally, the lifecycle of the JavaBean instances (corresponding to the Java Servlet specification's request, session, or application scopes) is configurable by the developer and managed by JSF. Developers do not need to explicitly manage their variables' life cycle.

One develops JSF applications by developing Web pages with HTML and JSF tag libraries. The code for processing the user input (the 'business logic') is implemented with JavaBeans, which are associated with JSF pages in a configuration file (faces-config.xml by default). A full description of JSF is beyond the scope of the current paper, so interested readers should consult McClanahan *et al.* [9]. However, the implications of JSF to science portal development are important. The immediate result is that we do not need to adopt HTML parameter naming conventions for our Web pages (and thus break our forms when we change names). More importantly, we can develop our Web application code as JavaBeans ('backing beans'), which are shielded from the Java Servlet specification. This allows us to reuse JavaBean code in non-JSF applications, take advantage of XML bean serialization tools, develop simple standalone unit tests, and generally take advantage of JavaBean-based 'Inversion of Control' [10] frameworks.

We have developed PWscf input pages with JSF to collect the user input needed to create a PWscf input page. A sample page of the VLab portal (http://pedro.msi.umn.edu:6080/gridsphere/gridsphere) is shown in Figure 1. Users must fill out two pages of forms to describe their problem with a chance to preview and (if PWscf experts) edit the generated input file manually before submission. Users may also upload additional input data (e.g. atomic pseudo-potentials) from their desktop. The linked input

DOI: 10.1002/cpe

pages and backing Java bean code together constitute a portlet. One of the issues we address is the persistent preservation of user input data. The form shown in Figure 1 is tedious to fill out, and quite typically a user will want to make minor modifications to a particular job and resubmit it later. This is part of the larger problem of metadata management, which has been investigated by projects such as the Storage Resource Broker [11] and Scientific Annotation Middleware [12]. For VLab, we are evaluating the use WS-Context [13], a lightweight, Web-Services-based metadata system. A 'context' is simply a URI-named collection of XML fragments. To support linked contexts in VLab, we have extended our WS-Context implementation to support parent–child relationships between contexts [14,15]. Context servers are normally used as lightweight metadata storage locations that can be accessed by multiple collaborating Web Services.

In our current work, the data collected from the user interface input form (Figure 1) is written into a unique context associated with that user session. This data is stored persistently using a MySQL database, although this implementation detail is not relevant to the PWscf portlet developer. Each user has a base context, which is subdivided into one child context per user session. These child contexts are used to store specific input parameter values for that particular job submission. These sessions may then later be browsed and the data recovered for subsequent job submission—the form in Figure 1 has its values repopulated.

Although we may store and recover values one at a time from the context storage, we are developing XML serialization techniques to more easily store and recover entire pages using Java bean serialization. Once stored in the WS-Context Server, the input forms can be unserialized and reconstructed on demand.

A serialized Java bean object may be stored and queried in WS-Context. According to the WS-Context specification, a Java object may be considered to be a 'context', i.e. metadata associated with a session. When storing a context, we first create a session in WS-Context Store. Here, a session can be considered an information holder; in other words, it is a directory where contexts with similar properties are stored. Each session directory may have associated metadata, called 'session directory metadata'. Session directory metadata describes the child and parent nodes of a session, and this enables the system to track the associations between sessions. One can create a hierarchical session tree where each branch can be used as an information holder for contexts with similar characteristics. These contexts are labeled with URIs, which give structured names to tree elements. For example, 'vlab://users/jdoe/session1' may refer to a session directory where contexts are stored and linked to a session name 'session1' and user name 'jdoe'. Upon receiving the system response to a request for session creation, the user can store the context associated to the unique session identifier assigned by the WS-Context Store. This enables the WS-Context Store to be queried for contexts associated to a session under consideration. Our WS-Context implementation normally allows for the specification of the lifetime of the metadata. For VLab, each context is stored with unlimited lifetime as the WS-Context Store is being used as an archival data store.

3. TASK MANAGEMENT

In the previous section we described the use of JSF to create form pages for collecting input data from the user. These input values are used to create an input file of the form expected by the PWscf application. We are now ready to connect with the Grid. Recall that we are using a three-tiered model

DOI: 10.1002/cpe

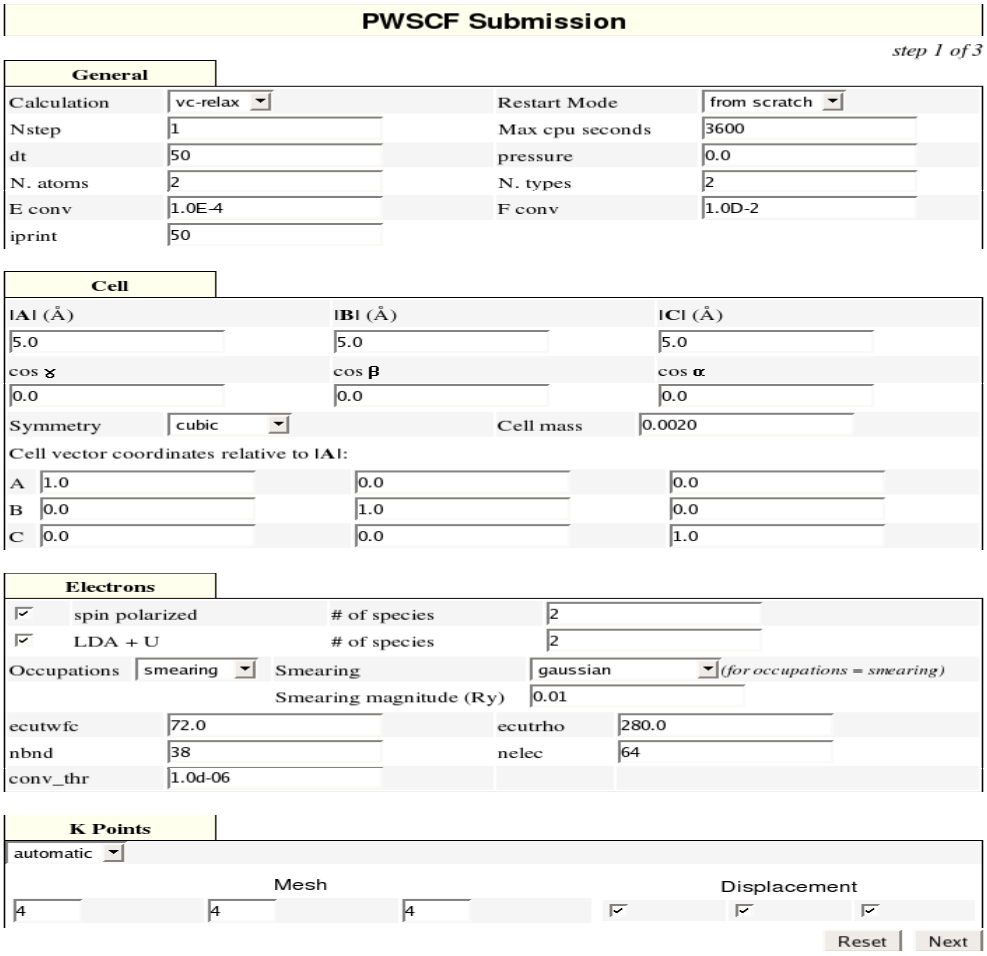


Figure 1. PWscf input forms are developed with JSF.

for our system: the portal server manages clients to Grid services, which in turn provides access to back-end computing resources. Our requirements at their simplest are as follows: (a) transfer the PWscf input file to the desired back-end resource (i.e. one with the PWscf executable installed on it); (b) invoke the PWscf application; (c) monitor the application; and (d) access the output data.

Many portals have provided these capabilities, and general purpose portlets for performing these tasks are available from the Open Grid Computing Environments (OGCE) project [16] and GridSphere [8]. Java-based Grid portals and portlets quite often are based on the Java CoG Kit [17], which provides a client programming interface for interacting with standard Grid services such

DOI: 10.1002/cpe

as GRAM, GridFTP, MyProxy, and Condor. More recently, the Java CoG has been significantly redesigned to provide abstraction layers for common tasks (job submission and remote file operations). These abstraction layers mask the differences between different Globus Toolkit versions and also support alternative tools such as job submission with Condor. The redesigned Java CoG also provides a way to group these tasks into workflow graphs that can accomplish sequences of operations. This is more thoroughly reviewed by Amin *et al.* [18,19].

Although existing portlets may be adapted to handle VLab tasks such as uploading input files to remote machines and invoking PWscf, this adaptation still involves a lot of work and does not encourage code reuse. One of our goals in this project, in collaboration with the OGCE, is to simplify Grid portlet development by using JSF tag libraries that wrap the Java CoG abstraction classes. This allows us to associate multiple actions with a single HTML button click. These actions can furthermore be grouped into composite tasks that correspond directly to Java CoG workflow graphs (see Amin *et al.* [18,19]).

However, JSF presents us with a problem, it only manages individual session beans. Furthermore, a user may need to submit many independent jobs within a single session, each with its own bean. Also, we must link several beans into compositions of multiple grid tasks—even the simple PWscf submission combines file transfer and job submission tasks into a single button click. The task and taskgraph managers described in this section represent our current solution to these problems.

The task manager handles independent user requests, or tasks, from the portlet client. The user request-generating objects are simply JavaBean class instances that wrap common Grid actions (launching remote commands, transferring data, performing remote file operations) using Java CoG classes. We define a general-purpose interface called *GenericGridBean*, which specifies the required get/set methods of the implementing class. These are data fields such as 'host name', 'toolkit provider', and other attributes needed by the underlying CoG classes. *GenericGridBean* implementations include *JobSubmitBean*, *FileTransferBean* and *FileOperationBean*. When a client invokes a particular type of action, it does so indirectly through the task manager, which is responsible for passing property values and calling action methods to the individual task beans. Once a user request is caught, the *TaskManagerBean* instantiates a *TaskBean* object and its event listener.

In addition to managing multiple independent tasks, we must also often manage coupled tasks: a single button click may require two or more actions that depend on one another. As described by Amin *et al.* [18,19], the Java CoG Kit provides the *TaskGraph* class to handle directed acyclic graph-like workflows composed of atomic tasks. As with the task manager previously, we have defined a *TaskGraphBean*, which is a JavaBean wrapper around the CoG *TaskGraph*. The taskgraph manager, which has an associated *FactoryBean*, coordinates user requests with *TaskGraph* backing beans. Each *TaskGraph* bean is itself composed of instances of *JobSubmit*, *FileOperation*, and *FileTransfer* beans.

The task and task managers store associated metadata to a persistent storage service, which in our implementation is a WS-Context Service. The storage service has methods that can access these bean instances with a unique key called 'taskname'. A JSF validator guarantees that each 'taskname' parameter is unique within the session scope. The task manager is responsible for monitoring individual tasks and managing their lifecycles. When a task is initially submitted, we store its property values and state in the Storage Service. Live objects correspond to the CoG states 'unsubmitted', 'submitted', and 'active'. When a task enters 'completed' or related states ('failed', 'canceled'), its submission, completion dates and output file(s) metadata are stored as well. This allows us to recover the submitted job's properties (such as the input file used and execution host) for later editing and resubmission.

DOI: 10.1002/cpe

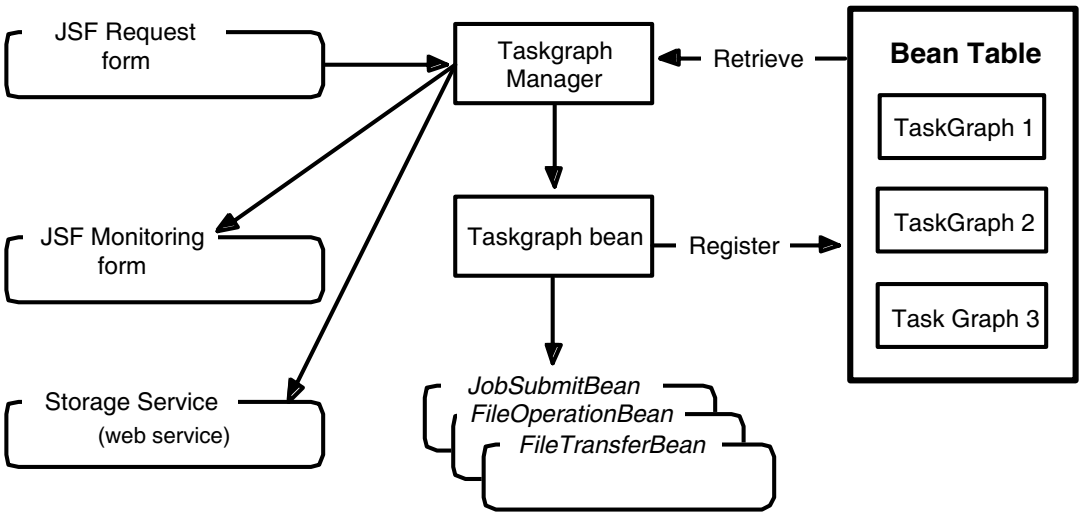


Figure 2. The taskgraph manager manages Grid submissions with multiple steps.

One current drawback to this initial scheme is that live objects are lost when the session expires. The Globus Toolkit services provide persistence with clients through callback ID handles, but this capability must be added to the Java CoG abstraction layer. Note that in the case of task graphs, we only store its metadata.

We are also investigating another solution to the persistence problem. One of the side effects of the JSF approach is that the JavaBean classes that encapsulate the Grid business logic may be developed independently of the JSF container. This will allow us to develop a purely Web Service version of our system, in which our Task Beans run as standalone Web Services. This approach will simplify the management of user objects that are not directly tied to Tomcat session objects.

We express the dependencies between tasks using JSF tag library extensions so that the JSF application developer can encode the composite taskgraph workflow out of reusable tags as shown in the JSF snippet below. When the user hits the 'submit button', the *TaskGraphBean* clones and submits itself, and finally registers itself to the bean table within the current session (Figure 2). The taskgraph manager has monitoring and event handling mechanisms. We may also use the taskgraph manager to retrieve specific *TaskGraph* instances so that we may, for example, check the current status of a running job. The JSF tag `<o:taskGraph>` tag represents a multi-staged task submission. In the current implementation, the `<o:taskGraph>` tag is composed of a file transfer task, followed by a job submission task, and another file transfer task. In addition, the user can specify a task to specify whether or not a serialized representation of the taskgraph is transferred to archival storage. Of course, the taskgraph can also be constructed from any subset of these. In the above example, not only is the taskgraph composed of three tasks, each task is dependent on the previous one. This dependency is explicitly stated through the `<o:taskAdd>` tag.

DOI: 10.1002/cpe

```
<o:taskGraph id="myGraph" binding="#{taskgraph.taskgr}" >
   <o:task id="task1" method="#{task.create}" type="FileTransferInput" />
   <o:task id="task2" method="#{task.create}" type="JobSubmit" />
   <o:task id="task3" method="#{task.create}" type="FileTransferOutput" />
   <o:taskAdd id="taskadd1" name="task1" depends="task2" />
   <o:taskAdd id="taskadd1" name="task2" depends="task3" />
   <o:contextStore id="context" type="taskgraph" />
</o:taskGraph>
```

In the example above, the first task transfers the input file from a remote location that corresponds to a GridFTP file transfer call. The second task executes a GRAM job submission call. The third task is another GridFTP file transfer to stage output file(s) to a remote archival or file server. The `<o:contextStore>` tag initiates the storage of taskgraph archival data onto the storage server.

4. COLLABORATIVE WEB SERVICES

The PWscf application represents a straightforward Grid application. VLab will also need to address more challenging problems: determining the best available back-end resources for scheduling applications and developing collaborative Web Services that allow multiple clients to interact with the same service. For example, the Task Manager described in Section 3 specifies specific host computers in the current implementation. Virtualizing the back-end connection through proxy services is highly desirable for both load balancing and fault tolerance. In this section, we explain our use of message-oriented middleware (NaradaBrokering) to investigate solutions to these problems. Note that we present a proof of concept only with a simple example.

The desired functionality ascribed to VLab demands a flexible environment that supports a variety of services related to computation, storage, visualization, database transactions, and processing. In addition, system scalability, expandability, and fault tolerance are essential components. We have chosen NaradaBrokering [2] as a middleware system that already incorporates many of these elements.

NaradaBrokering [2] is a Grid and Web-Service-compatible middleware framework formed by a cooperating set of brokers whose role is to process messages sent to it, each with a topic tag, and route them to any subscribers to that topic. The use of a topic tag is the hallmark of publish/subscribe systems, which make it possible to make requests and receive replies without regard to the specific system processing the requests. Using NaradaBrokering, we have developed a prototypical network, geographically distributed to illustrate the features of a future VLab system. The components of our architecture are illustrated in Figure 3.

To demonstrate the feasibility of our approach, we have developed a collaborative wavelet transform Web Service that uses NaradaBrokering to enable multiple participants to receive the same messages. Large-scale datasets are stored on two servers at the University of Minnesota. A client can request the wavelet transform of a specific dataset, and have a specified number of wavelet coefficients transmitted to the client. We implement the client as an applet (which can itself be embedded in a portlet) for acceptable interactivity and image rendering. However, the approach is general and can be applied also to other Web applications (i.e. JSF portlets described in Section 2). In our wavelet application, the applet displays the coefficients as spheres centered at the location occupied by the center of the 3D wavelet. High-performance graphics are obtained through the use of JOGL, an OpenGL API for Java.

DOI: 10.1002/cpe

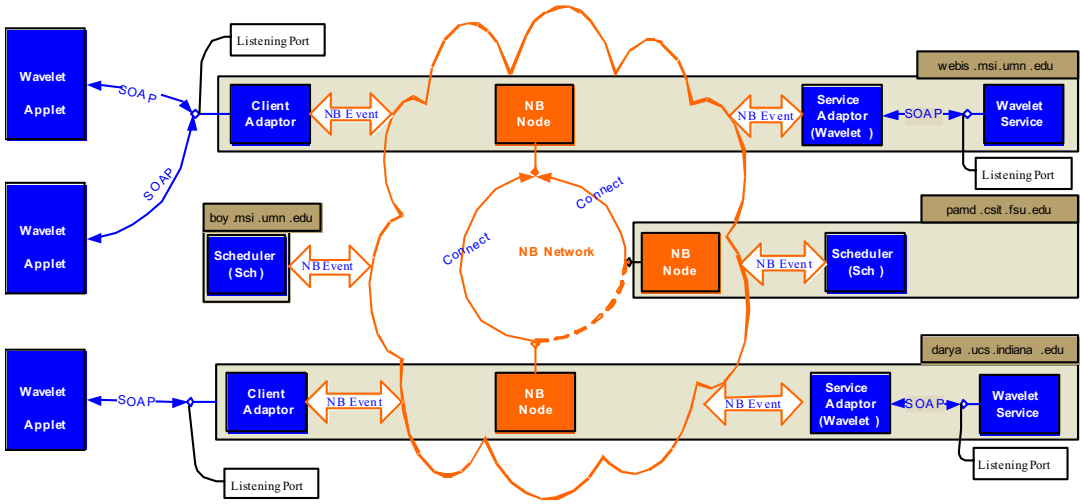


Figure 3. A network of three NaradaBrokering brokers is deployed across three sites. Attached to the network are two wavelet services (with service adaptors), two schedulers, and several client adaptors. Users access the wavelet service via collaborative applets.

Although we demonstrate our framework with a wavelet transformation service for visual impact, this service can be replaced with any other service that conforms to the WSDL standard. The system has several components that work together to isolate the user from the task of finding available services, finding the servers that support them, and distributing the workload. As shown in Figure 3, these components are the Broker Network, the schedulers, the services (with associated adaptors), the clients (with their adaptors), and the service registries (not shown). The Broker Network encompasses a collection of three NaradaBrokering nodes connected to one another. Any entity (client or service) publishing to any of these nodes will have its request propagated through the entire Broker Network towards one or several destinations. NaradaBrokering is responsible for efficient routing, guaranteed delivery, or in the event of problems, appropriate error feedback to the sender.

The Schedulers manage the task execution process. There are several schedulers to provide fault tolerance to the system, in the event that any of the servers that host the scheduler become unavailable. Different schedulers might also handle different types of requests. When a client makes a request, a scheduler is identified to handle the request and establish a connection between service and client. The services are responsible for completion of requests initiated either by the clients or by the schedulers on their behalf. Data generated from these services are either returned to the clients or to the schedulers who then forward the data to the clients. Each service adaptor is an interface between a service (which has no knowledge of the NaradaBrokering middleware and communicates via the SOAP protocol), and NaradaBrokering, which understands messages labeled with an associated topic. Thus, the adaptors wrap and unwrap messages received by and sent to the services, to make them conform to the NaradaBrokering protocol. To this end, the adaptors simply add a subscription or publishing topic,

DOI: 10.1002/cpe

and optionally adds some headers to the message, such as IP addresses. Services are responsible for executing the tasks generated by the clients. Clients may be applets, either standalone or within a portal environment, or portlets. They communicate with the middleware through the use of client adaptors, which play the same role with respect to clients as do the service adaptors with respect to services.

We now illustrate the path taken from an initial request by the client to display some imagery, to the receipt of the final image. This is built around standard topic-based publish/subscribe methods. The key concept within our system is that each entity (any one of the system's components with the exception of the brokers) is associated with a unique topic ID, and each entity subscribes to its own ID topic. This ensures that anytime a message is published to a topic that has the ID of one of the entities, only that entity will receive the message. Furthermore, each entity subscribes to a category topic ID related to the services it can provide. For example, a scheduler entity subscribes to the category ID 'scheduler', while the wavelet server might subscribe to the category ID 'wavelet'. These category IDs are not unique. There might be multiple services capable of handling schedulers or wavelet manipulation. In the first step, a client sends a message with the topic 'scheduler'. Through the publish/subscribe mechanisms, all entities that subscribed to 'scheduler' will respond, namely all the schedulers. Currently, the first scheduler to respond is chosen. If a particular scheduler is unavailable, the system does not break since a reply message will still be received from any available scheduler. The ID of the scheduler is included in its return message to the client, enabling the client to connect directly to it. The client then sends its specific task request (in this case, the request to perform the wavelet transform on a specific file) to the scheduler. The message headers include the name of the desired service (in this case the service has the topic 'wavelet'). The scheduler sends a message out requesting a wavelet service that has subscribed to the topic 'wavelet'. Once found, the wavelet service returns a message with its own unique ID. The scheduler will then either act as a proxy for the client, or the client will link the client to the service directly. We note that the original client task has its own ID associated with it, and any other client that uses that same ID will share its display with the original client. The registry entities provide the mechanism through which additional clients can enquire as to existing client interactions. In principle, authentication mechanisms can be used to determine if the client can connect to a particular session, but this was not implemented in our prototype.

5. CONCLUSIONS AND FUTURE WORK

The work described above represents the first steps in the development of a robust, collaborative system to provide simple, yet flexible, workflows for research scientists interested in conducting complex scientific computational tasks without having to worry about the intricacies of the underlying computational frameworks. To this end, we have addressed three important components of this framework: data entry, job submission, and back-end services. Our work is guided by the principle of ease of use, fault tolerance, collaboration, reusable code, and persistent records. The use of the PWscf code serves as a prototypical code, which forms a single cog in a more complex data entry, job submission, and data analysis cycle. More generally, there are several codes that must be submitted, often in large numbers, and there is often causality between results already generated and the codes to be submitted as a result of particular analyses. Implementation of a system that takes care of these dependencies (quasi-)automatically is an end objective of this project. To this end, we will focus our attention on several important components of the system.

DOI: 10.1002/cpe

Scheduling and Load Balancing. Although we include at least two entities of each type in our distributed system (2+ schedulers, 2+ wavelet services, etc.), there is as yet no attempt to take network and server load into account when choosing which units will perform the actual work. It is currently a first-come, first-chosen approach. We will evaluate existing work and enhance our schedulers and services to activate themselves based on a more realistic measure of instantaneous or extended load.

Collaboration. Collaboration is a natural attribute of our system. Two user tasks that subscribe to identical topics automatically receive the same information. We will investigate approaches to achieve this collaboration both at the visual level (shared user interfaces and displays), with the possibility of multiple users controlling the input. Much work has been done in this area, albeit (to the authors' knowledge) not within the context of publish/subscribe middleware. We have prototyped this system using a wavelet service and portal clients, as discussed in Section 5, but much additional work needs to be done.

Workflow. Complex workflows are important within VLab. Recent research has shown how to implement strategies for specifying workflows across multiple services. This work will be integrated within our system to properly link input, job submission, analysis, feedback to the user, and finally, automatic (or semi-automatic) decisions regarding the next set of simulations to submit.

In future work, the Web Service Resource Framework [6] and particularly the WS-Notification specification family may be used to replace our pre-Web-Service topic system. Support for WS-Notification is currently being developed in NaradaBrokering [20]. When this is available we will evaluate its use in our system.

ACKNOWLEDGEMENTS

This work is supported by the National Science Foundation's Information Technology Research (NSF grant Nos. ITR-0428774, 0427264, 0426867 VLab) and Middleware Initiative (NSF grant No. 0330613) programs.

REFERENCES

1. Virtual Laboratory for Earth and Planetary Materials Project Web site. http://www.VLab.msi.umn.edu/ [6 April 2007].
2. Pallickara S, Fox G. NaradaBrokering: A distributed middleware framework and architecture for enabling durable peer-to-peer Grids. *Proceedings of the 2003 ACM/IFIP/USENIX International Middleware Conference Middleware-2003*, Rio de Janeiro, Brazil, June 2003. Springer: Berlin, 2003.
3. Fox GC, Gannon D, Thomas M (eds.). *Concurrency and Computation: Practice and Experience* (*Special Issue on Grid Computing Environments*) 2002; **14**(13–15):1035–1600.
4. Democritos Scientific Software Web site. http://www.democritos.it/scientific.php [6 April 2007].
5. Plane-Wave Self Consistent Field Web site. http://www.pwscf.org/ [6 April 2007].
6. Foster I. Globus toolkit version 4: Software for service-oriented systems. *Proceedings of the IFIP International Conference on Network and Parallel Computing* (*Lecture Notes in Computer Science*, vol. 3779). Springer: Berlin, 2005; 2–13.
7. Abdelnur A, Chien E, Hepper S (eds.). Portlet Specification 1.0. http://jcp.org/aboutJava/communityprocess/review/jsr168/ [6 April 2007].
8. Novotny J, Russell M, Wehrens O. Gridsphere: A portal framework for building collaborations. *Concurrency and Computation: Practice and Experience* 2004; **16**(5):503–513.
9. McClanahan C, Burns E, Katain R (eds.). *Java Server Faces Specification 1.1*. Sun Microsystems: Sanata Clara, CA, 2004. Available at: http://java.sun.com/javaee/javaserverfaces/download.html [6 April 2007].
10. Johnson R. *Expert One-On-One J2EE Design and Development*. Wiley: Indianapolis, IN, 2003.

DOI: 10.1002/cpe

11. Rajasekar A, Wan M, Moore R, Schroeder W, Kremenek G, Jugatheesan A, Cowart C, Zhu B, Chen SY, Olschaowsky R. Storage resource broker—managing distributed data in a Grid. *Computer Society of India Journal (Special Issue on SAN)* 2003; **33**(4):4254. Other Storage Resource Broker publications available at: http://www.sdsc.edu/srb/Pappres/Pappres.html [6 April 2007].
12. Schwidder J, Talbott T, Myers JD. Bootstrapping to a semantic Grid. *Proceedings of the Semantic Infrastructure for Grid Computing Applications Workshop at the IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID 2005)*, Cardiff, U.K., 9–12 May 2005. IEEE Computer Scoiety Press: Los Alamitos, CA, 2005.
13. Bunting B, Chapman M, Hurlery O, Little M, Mischinkinky J, Newcomer E, Webber J, Swenson K. Web Services Context (WS-Context). http://www.arjuna.com/library/specs/ws_caf_1-0/WS-CTX.pdf [6 April 2007].
14. Aktas MS, Fox G, Pierce M. Managing Dynamic Metadata as Context, June 2005. http://grids.ucs.indiana.edu/ptliupages/publications/maktas_iccse05.pdf [6 April 2007].
15. Fault Tolerant High Performance Information Services Web site. http://www.opengrids.org [6 April 2007].
16. Alameda J *et al.* Open Grid Computing Environments collaboration: Portlets and services for science gateways. *Concurrency and Computation: Practice and Experience* 2007; **19**(6):921–942. DOI: 10.1002/cpe.1078.
17. von Laszewski G, Foster I, Gawor J, Lane P. A Java Commodity Grid Kit. *Concurrency and Computation: Practice and Experience* 2001; **13**(8–9):643–662. Available at: http:/www.cogkit.org/ [6 April 2007].
18. Amin K, Hategan M, von Laszewski G, Zaluzec NJ. Abstracting the Grid. *Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP'04)*. IEEE Computer Society Press: Los Alamitos, CA, 2004; 250–257.
19. Amin K, von Laszewski G. Java Cog Kit Abstractions. http://www.cogkit.org/release/4_1_2/manual/abstractions.pdf [6 April 2007].
20. Pallickara S, Gadgil H, Fox G. On the discovery of topics in distributed publish/subscribe systems. *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing (Grid 2005)*, Seattle, WA, November 2005. IEEE Press: Piscataway, NJ, 2005.