Building web services for scientific grid applications

G. Kandaswamy
L. Fang
Y. Huang
S. Shirasuna
S. Marru
D. Gannon

Web service architectures have gained popularity in recent years within the scientific grid research community. One reason for this is that web services allow software and services from various organizations to be combined easily to provide integrated and distributed applications. However, most applications developed and used by scientific communities are not web-service-oriented, and there is a growing need to integrate them into grid applications based on service-oriented architectures. In this paper, we describe a framework that allows scientists to provide a web service interface to their existing applications as web services without having to write extra code or modify their applications in any way. In addition, application providers do not need to be experts in web services standards, such as Web Services Description Language, Web Services Addressing, Web Services Security, or secure authorization, because the framework automatically generates these details. The framework also enables users to discover these application services, interact with them, and compose scientific workflows from the convenience of a grid portal.

Introduction

Large scientific collaborations are increasingly employing web-service-oriented architectures to manage their scientific enterprises. The reasons for this are tied to the fact that science is becoming more multidisciplinary and, to make progress on key scientific questions, it is increasingly dependent upon complex workflows of data analysis and simulation tasks. For example, in the area of biology, modeling a cell involves the integration of many complex, interacting processes, each of which may be understood by only one or two specialists on a team (physicists, biochemists, or complex systems engineers). In the area of severe storm prediction, as illustrated by the examples in this paper, a single storm prediction may involve the interaction of a data mining task with data integration from real-time instruments and massive simulations which, in the future, will drive the control of online instruments such as Doppler radars. These large research communities are often divided into teams of specialists, each responsible for a particular set of resources and applications. All of the resources used to host each team's applications—together with a layer of services (described below) that ties them together—are often referred to as a grid. The users of this grid are the members of the community who need to invoke these applications individually or as part of a workflow to solve particular problems. To build these workflows, it is necessary to overcome the barriers introduced by the heterogeneous nature of the community's grid and applications. Web services are designed specifically to solve this problem, and they have become the foundation for most new grid standards.

Unfortunately, most of the applications developed and used by scientific communities are command-line applications written in Fortran, C, and a host of scripting languages. These applications are fast and efficient, and they represent state-of-the-art science; however, they are often platform-dependent and difficult to integrate with applications from other disciplines. Programmatic access to these applications from remote clients is usually difficult. Many of them lack a graphical user interface (GUI), which makes them cumbersome for end users. Also, there is no standard way to describe their input parameters and output results or to monitor their

0018-8646/06/$5.00 © 2006 IBM

progress as they run for extended periods of time on the grid. By converting these command-line applications into application services, we can overcome many of the limitations mentioned above. In this paper, an *application service* is an application with a web-service interface that is described by the Web Service Definition Language (WSDL) [1] as a set of endpoints operating on messages containing document-oriented information.

A web service can *wrap* an application, enclosing it and invoking it without the application having to be modified. In principle, the task of wrapping an application as a web service is not difficult for a specialist trained in web and web-service programming, but for most scientific application specialists, this is an extremely high barrier to surmount. There are a number of tools to help accomplish this task. However, simply providing a web-service interface is not sufficient to make the application a usable component in a distributed computation. One major concern is security. In particular, how can a scientist provide a selected group of users with access to application services without building a separate security infrastructure or requiring users to have login accounts on the host running the service? Another problem is making the service usable directly from a web portal, as well as making it a component in a workflow. The primary research contribution of this paper is to describe a framework we have built that allows scientists to wrap their applications as web services and deploy them on a grid; it also automatically provides an authorization system that allows selected users to securely interact with these services through automatically generated web interfaces, to compose scientific workflows using these services, and to monitor the status of their workflows on the grid using standard publish-subscription notification systems. The framework has four primary components:

- *Grid portal*—A web server and gateway with which users may access services, compose workflows, and manage data. The portal is used by the application provider to create the service for others to use and by the users who wish to interact with the service through its automatically generated web interface.
- *Generic factory service*—Invoked from the portal by application providers to wrap applications as services and create new instances of these services on the grid (first introduced in [2]). It is integrated with a capability-based authorization system that allows fine-grained control over user access to the deployed services.
- *Workflow composer*—A tool that allows users to compose complex and interesting workflows from application services.
- *Notification service*—Allows application services to send messages that are logged by the portal and monitored by the workflow instance.

Finally, there is the grid itself. There are perhaps as many definitions of the term *grid* as there are grid deployments. The Global Grid Forum is in the process of defining a reference model for grids called the *Open Grid Service Architecture* [3]. However, for the tools described here, we assume very little. We define a grid to be a collection of computers that are configured in such a way that if they are separated by firewalls, a range of Internet Protocol (IP) ports have been left open so that services hosted on these machines can exchange messages. In addition, we assume that at least one machine in this set has some ports open to the outside world so that external facing services, such as the portal, can be used by clients elsewhere. We also make extensive use of the grid security infrastructure (GSI) security model [4] for user authentication. GSI offers us two advantages:

- A user authentication framework that allows an application provider to present a signed certificate of identity that authenticates that user with each resource in the grid. This is called the *single sign-on* property because a user needs to enter a password only once to obtain a proxy certificate, which can be used by the portal or other clients to act on behalf of the user.
- A mechanism that allows remote execution, so that a process with a valid certificate running on one grid resource can instantiate a process on another grid resource. This can be done using the GSI-enabled SSH (Secure Shell) protocol or the Globus Resource Allocation Manager (GRAM) protocol.

In general, installing Globus [5] is one easy way to build a grid, but there are other solutions, such as the Open Middleware Infrastructure Institute [6] framework that may also be suitable. Within the grid, we deploy a set of persistent services that is part of the framework that supports the application services we create. These include the grid portal server, which contains the generic factory service used to create instances of application services (**Figure 1**).

The portal server also has a database that stores user context information that is loaded into the user session when the user connects through a web browser. The portal also loads the user's proxy certificate from a certificate vault and service authorization capabilities via a capability manager. When an application provider uses the factory service to start an instance of an application service, the application service registers its WSDL with a persistent service registry. The application service also publishes notifications about itself to the persistent notification service. To support service clients that are outside the grid and may not have fixed IP addresses, a message box service is provided as a messaging proxy. External clients can route messages to other services through the message box, or they can pull event notifications from a queue maintained on their behalf by the message box.

Related work

Over the last several years there has been substantial progress in building grid applications by composing them from predefined components and web services. The most notable among these are Taverna [7], Triana [8], and Kepler [9]. Each of these represents a powerful tool for composing workflows. Taverna is part of the myGrid project, focused on building middleware to support data-intensive experiments in molecular biology. Taverna has more than a thousand services that can be used as components in workflows. It also has an elegant approach to service discovery and capturing the full metadata context, including the provenance of all aspects of the scientific experiment represented by the workflow, including the data derivations and the workflow audit trail of invoked services.

At the user level, Triana provides an elegant and well-tested composition tool and a large toolbox of ready-to-use components. For grid application, Triana uses a software layer, called the *grid application prototype*, to distribute subsystems of the workflow graph to remote grid resources for execution.

The approach Kepler takes is based on an actor-oriented model that allows hierarchical modeling and dataflow semantics. The Kepler tools support a well-designed graphical composition interface that is very intuitive and easy to use. To support the interaction with web services, Kepler uses a form of actor proxy for each web service that is invoked.

While these and other frameworks have been developed to compose and run scientific workflows on a grid, few have addressed the issues of security, and most of them do not support wrapping an application as a web service. Soaplab [10, 11] is a set of web services that provides programmatic access to some applications on remote computers. It can create two types of web service—analysis service and derived analysis service. While the former allows users to send input data as weakly typed name–value pairs, the latter has strongly typed methods for sending input data and receiving results. Soaplab uses Apache Axis [12] to create Sun Java** implementation classes and deployment descriptors for all derived analysis services. It uses CORBA [13] on the server side for finding, starting, controlling, and using applications.

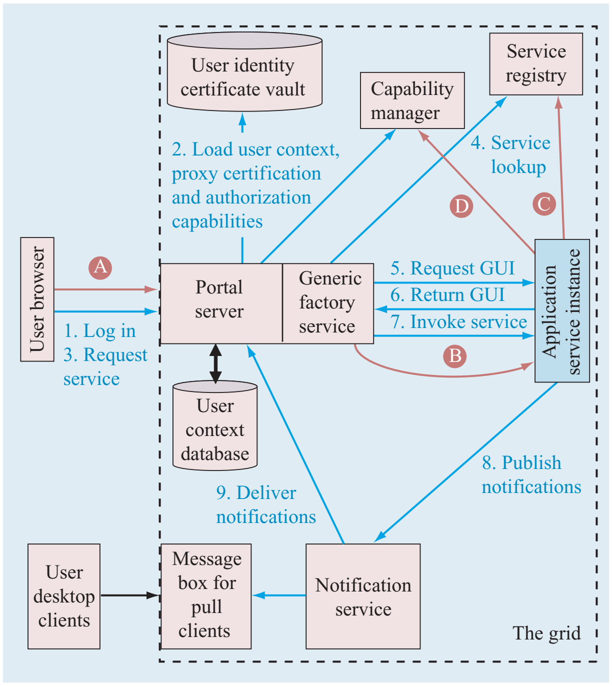


Figure 1

Scenario for creating and using an application service.

Although Soaplab serves to wrap as a web service almost any command-line tool, it has a number of limitations. Soaplab does not have a web-service-based notification system that can accept CORBA events and propagate them to clients, but it does implement the Object Management Group life-science analysis engine, which has a CORBA-based event notification model. Soaplab does not support grid standards for service-level authentication and authorization.

Gowlab [14] is an application that enables ordinary web pages to be wrapped as web services. It also allows programmatic access to these services. However, these services are difficult to maintain because of the nonstandardized and changeable nature of web pages. Also, most web pages are nontrivial and require the Gowlab service provider to write Java implementation classes to extract information from them.

The GridLab [15] project aims to provide application tools and middleware for grid environments. It uses the grid application toolkit (GAT) [16] in Triana, which is a set of application program interfaces (APIs) that grid application programmers can use for uniformly accessing numerous grid services and middleware. However, GAT does not address the problem of wrapping existing applications as web services.

There are a number of other frameworks, such as SeqHound [17], BioMOBY [18], and KEGG [19], that are specific to bio-informatics and may not be general enough for applications in other domains.

Building services from applications

The generic service toolkit described here allows application providers to wrap their applications as secure web services. There are two primary components to the toolkit:

- The *generic factory service*, which is a stateless and persistent web service that can be used for creating and instantiating application services.
- The *generic service client*, which can be used to interact with the factory and the application services that it creates.

To understand how the factory service can be used to create application services, we describe a typical use case. Suppose there is an application MyProgram that runs from the command line with a single integer parameter. Suppose also that MyProgram expects to find an input file `inFile` and produces an output file `outFile`. The user would like to create a web service that will allow anyone known to the web portal as a member of the MyTrustedUsers group to run this program. The user should provide only the value of the input parameter, the location of the input file, and the location in which the output is to be stored. The user should be able to provide this information either from the portal through a web interface generated by the application service or as part of a workflow. We assume that the input files are large and that they may be stored in some remote location, which is the case in many scientific enterprises. Consequently, we do not assume that the user will upload the file directly to the portal or the service, though this is possible. Rather, it is often better to pass the Uniform Resource Locator (URL) for the input files to the service. On the grid, this is often a GridFTP protocol [20] URL or a regular file path if the resources are network-file-system-mounted. The user should also receive a notification when the task has been completed or has failed.

We assume that MyProgram has been installed on some host. For example, assume that it is at *MyHost.com* and is located in the directory path */u/myapps/bin/MyProgram*. There are two actions that our application provider must take to create an application service with the above capabilities:

- Write a service map document (SMD), which is a simple Extensible Markup Language (XML) description of the service. We describe the document for this case in detail below. The document contains information about the parameters of the application, the location of the application, and the policy information concerning which individuals and groups are authorized to invoke the service.
- If the application is to be run on a machine that is part of a Globus-based grid, the service can be instantiated from the portal. In this case, the application provider logs in to the portal and uses the generic service portlet to upload the SMD. We describe this step in greater detail below. If the application is on a machine that is not on a Globus-based grid, it is possible to start the service from the command line on that machine.

The letters in Figure 1 illustrate the detailed flow of service interactions. In step A, the application provider has uploaded the service map document to the portal. The portal then pushes the SMD to the factory service, which, in step B, uses GRAM to launch the application service on the remote host. The application service on the remote host uses the service map to configure itself as the web service and generates its WSDL, which is registered with a well-known registry service so that it can be discovered by the portal or a workflow (step C). Finally, in step D, the service registers the authorization capabilities with a capability manager. The details of the capability manager are described in greater detail in the next section of this paper.

The numbered steps in Figure 1 illustrate the user view of an interaction with the service. When a user logs in to the portal, it loads the user's authorization capabilities. Then, using the generic service portlet, the user can search the service registry for the service and, in step 5, request the user interface to the service. The interface, which is generated by the service and returned to the portal, provides the user with a web interface to supply the input parameters to the service. If the user has the required authorization capabilities, the portal automatically invokes the service. In step 7, the service invokes the application and passes it to the parameters defined in the SMD. The service runs the application and publishes notifications about the progress of the execution.

Service map schema

The service map schema contains information about the service port types, including their operation and input and output parameters. It also contains configuration parameters that are needed to instantiate the service. An SMD that conforms to the service map schema has three main elements: service, port type, and creation parameter. The *service element* describes the service to be created. It contains the name of the service and a short description of the service. It also contains metadata about the service. In our simple example of a MyProgram service, the document has the top-level structure shown below. Note that the document is XML, but we have tried to present it in a more human-readable form that uses indentation to denote nesting and italics to represent string values. (Our future work includes building a tool to guide users through the construction of the document without requiring the users to write raw XML.)

```
ServiceMap:
  service:
    serviceName: MyProgram Service
    serviceDescription:
      This is the service wrapper to execute
      MyProgram.
  portType:
    Method: ...
    Method: ...
  creationParameter: ...
```

The *port-type element* contains a list of operations (also known as methods). Each operation has a name, an access policy, a description, metadata, and, if desired, a list of default values. The main method in our service example is "run," which is the one that we use to pass the application parameters *inFile* and the integer we call *count*. The complete specification of this method is shown below. There are several things to notice. First, the run method has an associated application tag that provides a path to the application. The tag policy defines the authorization policy. In the current implementation, the policy specification is quite simple and allows the application provider to specify a list of users and groups authorized to access the service and a lifetime for the policy. In this case, it indicates that users in the group defined by *myFriends* are allowed to invoke this method.

Each parameter can also have associated metadata, which can be used by the portal to help guide the user in the selection of a valid parameter. In the case of the output file URL, the portal can use the metadata to present the user with a set of valid locations that the user and the service can access and store the output file. In this case, the metadata for the input parameter *outFile* specifies a requirement that the directory must be accessible by GridFTP. This information allows our workflow composer to query the registry service for input parameter values that satisfy the input parameter requirements. In the future, it will allow the workflow composer to compose workflows by allowing users to connect outputs of a service to the inputs of other services only if they are semantically correct. Finally, there is an output parameter called *outFileURL*, which is a message generated by the service when the execution terminates. Output parameters are used when the service is employed in a workflow and the user wishes to use the output of this service as input to another.

```
Method:
  methodName: Run
  methodDescription: Run myApplication
  policy:
    group: myFriends
  application (type="jython"):
    path: scripts/myApp-script.py
  inputParameter:
    parameterName: inFile
      parameterDescription:
        The GridFTP URL of the user's input file
  inputParameter:
    parameterName: count
    parameterDescription:
      the integer command line parameter for the app
  inputParameter:
    parameterName: outFile
    parameterDescription:
      The GridFTP URL of the directory for where the
      user wants the output file stored
    metadata:
      nameValuePair:
        name: AccessServiceType
        value: GridFTP
    outputParameter:
      parameterName: outFileURL
      parameterDescription:
        This is an output message that can be used as
        an input to another service in a workflow
```

In the run method, there are two additional parameters that are hidden (i.e., they are values that are not provided by the user). These values are used by the portal and the workflow to identify the URL of the message broker that is the target of notifications. The second is the topic of the notifications. The topic uniquely identifies this service or workflow invocation so that it is possible to subscribe to the message stream to see messages related only to this service invocation.

```
inputParameter (hidden="true"):
  parameterName: brokerURL
    parameterDescription:
      This is the URL of the broker to which
      messages will be sent
  inputParameter (hidden="true"):
    parameterName: topic
    parameterDescription:
      The topic to which notification messages
      will be sent
```

The *creation parameter element* specifies the physical location in which the service will be created and the other parameters that are needed to create the service, including the installation path for the service code base, the application working directory, a temporary directory for the application to save temporary files, and a range of ports to use for the service.

```
creationParameter:
  host: myHost.com
  gFacPath: /usr/local/gfac
  workDir: /u/myapps
  tmpDir: /tmp
  java: /l/jdk1.4/bin/java
  portRange:
    start: 12346
    end: 12446
```

Creating and accessing services from the portal

The standard grid portals we describe here are based on the JSR-168 portlet container model [21] as implemented in the Open Grid Computing Environment portal framework [22]. When a user authenticates with the portal, a context is created for that user session. What the user sees is a set of portlets that each have a user interface and some back-end logic that runs in the portal server. The application provider uses the generic factory service to wrap an application as a web service. To do so, the provider first uses the proxy-manager portlet to load the grid proxy certificate into the portal context. The provider then uses the generic service client (also known as the *generic service portlet*) to access the factory. To create a service, the SMD is uploaded to the portlet, which then transfers it to the factory. After validating the service map document, the factory creates and starts the service on the specified host using GRAM [23, 24]. After it is instantiated, the service registers its WSDL with a registry service. This allows the service to be discovered by interested clients and end users who search for it in the registry using its name or metadata. The registry returns the service WSDL that the clients use to access the service. While client programs such as workflows access the service programmatically, end users rely on a web interface that is generated by the service itself—a concept that we borrowed from the Web Services for Remote Portlets specification [25].

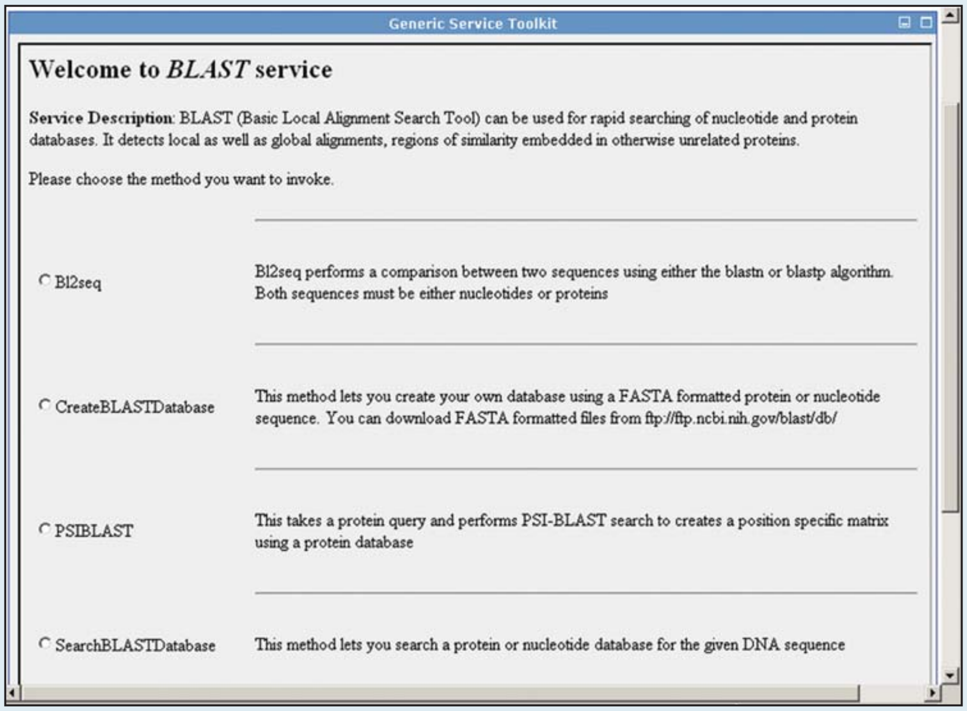
When a user accesses a service using the generic service portlet, the portlet requests the service for a user interface. The service then creates a user interface, which is a Hypertext Markup Language (HTML) form, and sends it back to the portlet, which is displayed to the user. The user interface shows all of the operations that the user is allowed to invoke on the service. When the user selects an operation, the portlet sends another request to the service to obtain the user interface for that operation—an HTML form that the user must complete in order to invoke the operation. In the SMD, the application provider can describe the user interface the service should create for its users.

For example, the application provider can specify the user interface for an input parameter using its *displayAs* attribute, which tells the service what HTML form element to use to display the default values for that input parameter. A number of values are supported for this attribute, including ListBox, RadioButton, CheckBox, and RemoteFile. Some of these have special properties; for example, the RemoteFile provides a browse button for the user to upload a file to the portlet, which then transfers it to the application service using GridFTP.

For example, **Figure 2** illustrates the user interface generated from the SMD for the standard basic local alignment search tool (BLAST) service [26]. In this case, there are four methods that can be invoked. The text in the web page is generated from the description of the methods in the service map.

When the user selects a specific method (in this case, B12seq), the user is presented with the specific parameter form that must be completed to invoke that method (**Figure 3**). After the user specifies all input parameter values for an operation, the portlet sends a Simple Object Access Protocol (SOAP) message to the service to invoke that operation. The service then invokes the application and sends notification messages about its status and the status of its application to a notification service. Clients and end users can then listen to these notification messages by subscribing to the topic to which the notification messages are being sent.

The notification service we use is called Web Services Messenger (WS-Messenger). It is our implementation of a notification model and is compliant with the WS-Notification [27] and WS-Eventing [28] specifications. It provides a messaging service for web services based on the publish and subscribe paradigm. WS-Messenger uses a topic-based notification channel for sending notifications. A topic is a subject of common interest among the services participating in a workflow to which all notification messages of the workflow are sent. The publish and subscribe notification mechanism and the creation of a topic are transparent to end users. There are three main components in our notification model: the notification consumer, the notification publisher, and the notification broker [29]. The notification consumer is an event sink; it is a web service that waits for notifications to arrive and handles them appropriately. All services that have to receive notifications use this service. Some clients, such as desktop tools, must listen for notifications that are generated by services. Unfortunately, these clients are often not associated with a fixed IP address that is visible to WS-Messenger, or they are behind a firewall. Consequently, we use a subscription proxy tool called the *message box* [30] that can hold message



Welcome to *BLAST* service

Service Description: BLAST (Basic Local Alignment Search Tool) can be used for rapid searching of nucleotide and protein databases. It detects local as well as global alignments, regions of similarity embedded in otherwise unrelated proteins.

Please choose the method you want to invoke.

○ Bl2seq	Bl2seq performs a comparison between two sequences using either the blastn or blastp algorithm. Both sequences must be either nucleotides or proteins
○ CreateBLASTDatabase	This method lets you create your own database using a FASTA formatted protein or nucleotide sequence. You can download FASTA formatted files from ftp://ftp.ncbi.nih.gov/blast/db/
○ PSIBLAST	This takes a protein query and performs PSI-BLAST search to creates a position specific matrix using a protein database
○ SearchBLASTDatabase	This method lets you search a protein or nucleotide database for the given DNA sequence

Figure 2

BLAST service interface generated from the service map document (SMD).

subscriptions for mobile clients or those hidden behind firewalls. These clients periodically pull messages from the message box rather than having them delivered via a standard WS-Eventing push.

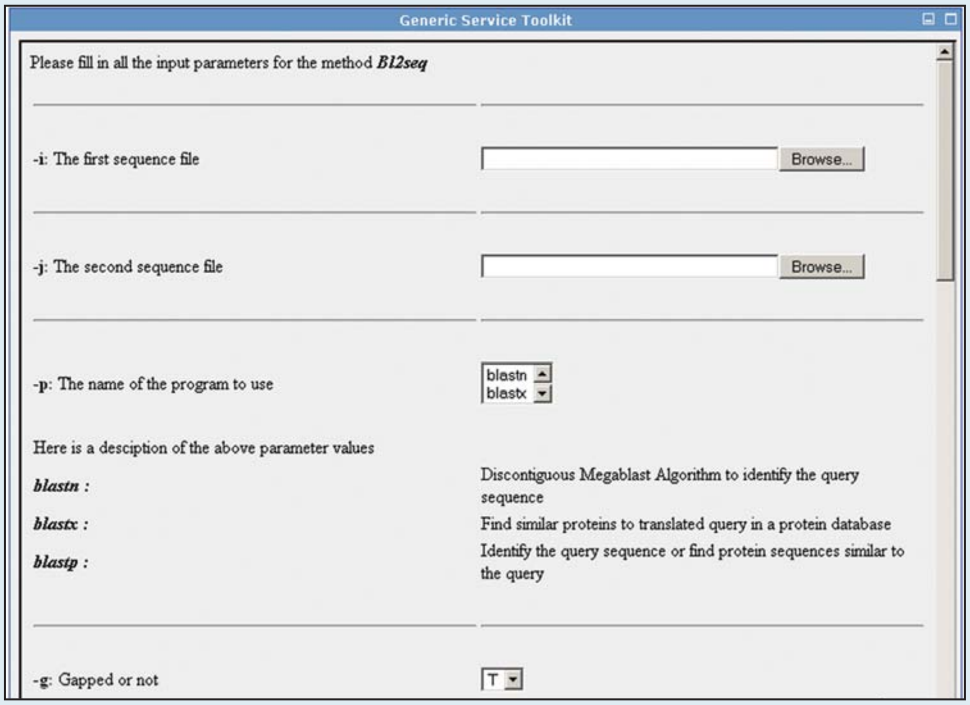
The workflow execution engine that executes a workflow subscribes to this service to receive notifications from all of the services in the workflow; the notification publisher is used to publish these notifications. All services created by the generic factory service have a built-in notification publisher to publish notifications. The notification broker is a service that acts as an intermediary and relays messages from a publisher to a consumer. It is persistent in nature and stores messages that cannot be delivered to the consumer. These messages can be retrieved later by the consumer.

A mediation approach is used to achieve compliance between the WS-Notification and WS-Eventing specifications. The WS-Messenger broker automatically converts the messages from one format to the other according to predefined transformation rules. The subscription request type for a listener determines the message format that the notification consumer receives. If a WS-Notification subscription request is received by the broker, it sends WS-Notification messages to the listener. Similarly, if a WS-Eventing subscription is received by the broker, the broker sends WS-Eventing messages to that listener. The publisher can publish messages in either format to the WS-Messenger broker; it makes no difference to the notification consumers.

Architecture of an application service

The factory creates an application service from its description in the SMD. Neither the application provider nor the factory generates any code for implementing the service interface. Also, no client-side stubs or server-side skeletons are created. (A stub is a client's local proxy for a remote object. The client uses the stub to communicate with the remote object—actually, the skeleton of the remote object. The skeleton is responsible for dispatching the client's communication to the actual remote object.) So how does the factory convert a service description into an actual service?

The answer lies in the *message processor* that is present in all application services created by the factory. The message processor is a very simple web service. It receives SOAP request messages and returns SOAP response messages, as many web services do. However, it cannot process the request messages because it does not know



Please fill in all the input parameters for the method ***Bl2seq***

-i: The first sequence file	
-j: The second sequence file	
-p: The name of the program to use	blastn
Here is a desciption of the above parameter values	
blastn :	Discontiguous Megablast Algorithm to identify the query sequence
blastx :	Find similar proteins to translated query in a protein database
blastp :	Identify the query sequence or find protein sequences similar to the query
-g: Gapped or not	T

Figure 3

User interface for the Bl2seq method generated by the BLAST service.

how to process them. The information that it needs to process the request messages is given to it by the application provider as an SMD. When given an SMD, the message processor reconfigures itself to support all the operations specified in the document. It then generates its WSDL and registers it with a registry. On the basis of the information in the service WSDL, a client can create a SOAP request message for the operation that the user wants to invoke on the service. When the message processor receives such a request message, it validates the message and invokes the application associated with the operation in a separate thread of execution. The mapping between the operation and the application is obtained from the SMD. The list of input parameter values for the operation has a one-to-one correspondence to the list of input parameter values for the application. The factory can invoke almost any command-line application, such as UNIX** commands, or more sophisticated scripts written in languages such as Jython, Python, and Perl. The design of the message processor is simple yet powerful. It makes the application service lightweight but highly configurable. No code generation is needed to create a service from its description.

Security in application services

There are two primary security components in this system: authentication and authorization. Our working assumption is that users authenticate with the portal server through a standard https-based username and password or, if available, a more advanced approach. The portal loads a grid proxy certificate into the user's portal context. The proxy certificate is an X.509 certificate extension defined in the GSI for the purposes of delegation and single sign-on.

Once authenticated, the portal fetches capability authorization tokens from the authorization system. Our authorization system, called *XPOLA*, is a fine-grained authorization infrastructure for web and grid services based on capabilities [31] and the principle of least authority (POLA). A capability token is a detailed policy document containing authorization information for each service instance and all of its operations. It is signed by the application provider proxy certificate. The XPOLA infrastructure includes a persistent capability manager service, plug-in capability handlers on the service and client sides, and a portlet-based user interface for application providers and users to manage their capability tokens. Enforcement plug-in handlers ensure that a user request can do no more than what is allowed as specified in the assigned capability token. XPOLA also supports role-based authorization by providing a group manager service. A service provider may create groups from users' distinguished names or make use of the existing group definitions to create supergroups from groups. The capability manager service and group manager service have persistent support based on a database. It is important to note that application providers and users rarely need to manage their capability tokens because the framework manages them on their behalf, as follows:

- The application services create the capability tokens from the policy information in the SMD and register them with the capability manager service.
- The generic service portlet contacts the capability manager service and loads the user's capability tokens into the user's portal context just before accessing any service.
- The application services renew their capability tokens before they expire. The renewal policy is specified in the SMD by the application provider.

In the previous section, we outlined the basic protocol to access a service. We now describe it in more detail with respect to security. To access a service, a user logs in to the portal and then uses the generic service portlet to access the service. The portlet contacts the capability manager service and loads the user's capability tokens into the portal context. The portlet then sends a request to the service using the secure socket layer (SSL). The user's proxy certificate is used for authenticating the user to the service, and the application provider's proxy certificate is used for authenticating the service to the user. After mutual authentication, the service returns the user interface. All further interactions with the service are secure and are done through the portlet using SSL. When the user invokes an operation on the service, the portlet sends a SOAP request message and the user's capability token, signed by the user's proxy certificate. The service verifies the authenticity of the request and, if the user is authorized to do so, carries out the operation according to the capability token. While this protocol may seem potentially complex, the framework handles the complexity in a manner that is transparent to application providers and users. A more detailed and quantitative analysis of the performance is underway and will be published later.

Our web services rely on proxy certificates to authenticate users. A proxy certificate has a short lifetime of two hours to minimize the damage if it is compromised. However, if a service must run longer than that, the proxy certificate must be renewed. The certificate is renewed by the application service, which contacts a MyProxy server that stores the application providers' credentials [32].

Composing workflows from the portal

So far, we have described how to wrap a single application as a web service. In this section we describe how to create workflows from the services using our workflow composer, called *X-Workflow*. X-Workflow provides an easy-to-use GUI that allows users to search for interesting services, visually connect them together to form workflows, and execute the workflows on the grid. Although this is not a new concept, X-Workflow has been designed with some additional features. It can monitor the progress of a workflow by subscribing to notifications from the workflow, make use of the metadata in the SMD to compose workflows, and compile the workflow into a standard workflow language for execution.

As discussed in the building services section above, an application provider uses the SMD to describe a service. This document is converted to an abstract WSDL by X-Workflow and registered with a registry. While a WSDL represents a service instance, an abstract WSDL represents a service. It allows the user to create workflows from nonexistent services. These services can be instantiated dynamically by the factory when the workflow is actually executed on the grid. The abstract WSDL contains information about the port types of the service, the operations, and their input and output data types.

Using X-Workflow, the user first searches a registry for interesting services. Each service is represented as a node with one or more inputs and outputs. The user creates a graph by interconnecting the services that constitute the workflow. The abstract WSDL also contains metadata about the service, port types, operations, and input and output parameters. Some of this metadata is made available by the application provider in the SMD. Other metadata about the input and output parameters can be obtained from other sources, such as the THREDDS [33] catalog generator service. After the user creates the graph that represents the workflow, the composer analyzes the dependencies among the constituent web services. It can then compile the workflow into a Jython script or a BPEL [34] script.

A particular abstract application service may have several concrete instances running at the same time on the grid. Each service instance may have a different policy associated with it. For example, services of a particular scientific community may not allow users of other communities to access them. The services created by our generic factory service include this policy information in the metadata for the operation elements in the WSDL. Before executing the workflow, our workflow engine

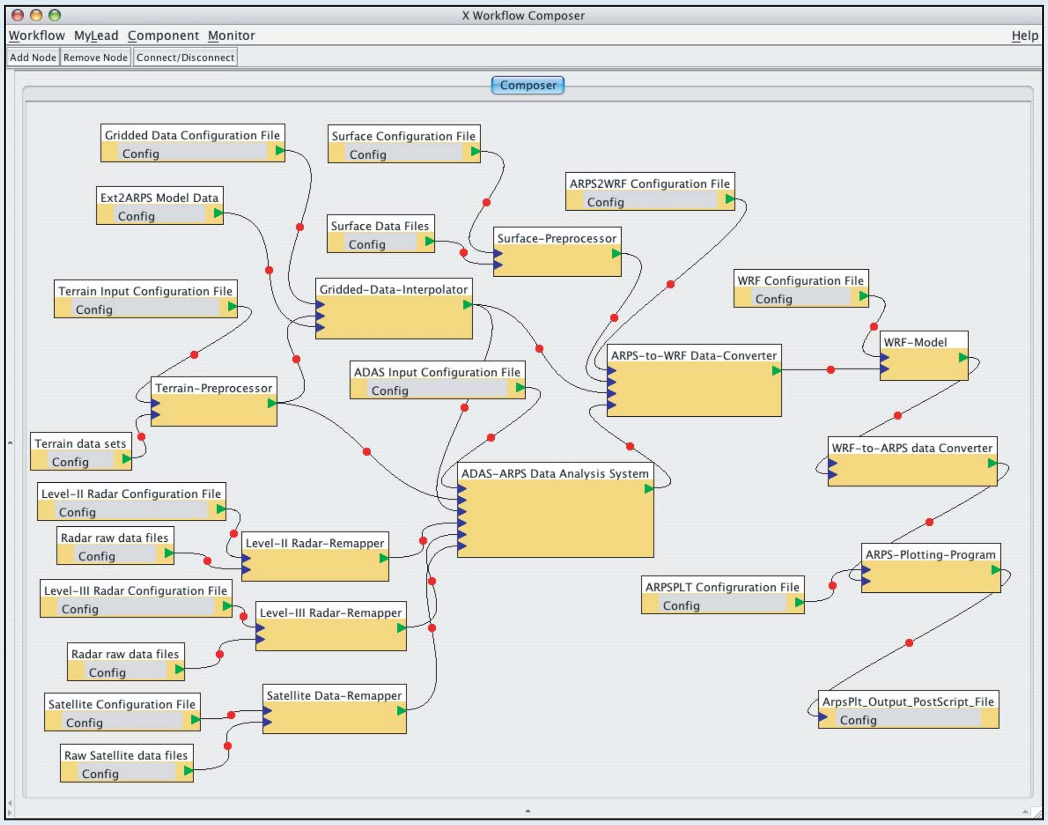


Figure 4

Atmospheric Data Assimilation System–Weather Research Forecast system workflow.

searches the registry for service instances that satisfy the policy requirements. After starting the workflow, the workflow engine receives notification messages from the services to monitor their status. Notification messages contain status information and output parameters. The workflow engine may use the output parameters of a service as the input parameters to the next service in the workflow. It also monitors the services for error messages. In the event of a failure of a service invocation, the entire workflow is stopped.

Example application

We have successfully used our generic service toolkit to wrap scientific applications as web services for the Linked Environments for Atmospheric Discovery (LEAD) project [35]—a large National Science Foundation Information Technology Research (NSF ITR) grant to advance the art of severe storm prediction using advanced grid and supercomputing methods. In this section, we show some of the services we have created and how they are used by X-Workflow to compose workflows for weather forecasting.

Figure 4 is a screen shot of the sample LEAD-coupled mesoscale weather forecasting experiment, which consists of prediction initialized by the ARPS Data Analysis System (ADAS) [36, 37] and forecasted by the Weather Research Forecast (WRF) model [38] system. The workflow involves 11 distinct services associated with 16 different input and configuration files. The application services are all wrapped with the tools described here.

This workflow is not yet in service as a production weather forecast tool, but it is being evaluated with the help of the LEAD meteorology team. One of the goals of the LEAD project is to put this, and other similar workflows, into the hands of forecasters and researchers as part of the Storm Prediction Centers *Spring Experiments 2006*, which tests promising new meteorological insights and technologies with the goal of creating advances in operational forecasting techniques for hazardous mesoscale weather events. To accomplish this goal will require substantial testing and, we are sure, countless improvements to the implementation of this service architecture.

Conclusions

Our generic framework allows scientists to wrap applications as web services, compose workflows from them, and execute them on a grid. The services created by our framework generate their own web interface, which allows end users to interact with them using thin and generic web service clients. Security is one of the prime concerns in a distributed environment. Our framework manages authentication and authorization during all interactions between clients and services in a manner that is almost transparent to application developers and end users. Users can search for services and compose workflows by visually interconnecting them in a workflow composer. Services use notification messages to report their status and results.

Much more work is needed to understand how well this system works in real applications. We are currently using this system with the LEAD severe storm prediction project and a bioinformatics grid project, but we have not done a rigorous usability analysis. We are currently planning to build a tool to allow users to compose SMDs without having to type XML. One thing we have learned is that complex workflows involving many services may require a complex array of input configuration and data files. Users may want to edit these files or interact with them in other ways. We need to be able to generate a good user interface for workflows in the same way that we can generate interfaces for services. Finally, we need to evaluate the robustness of the services and make them scalable by automatically creating new instances when the demand is high.

Acknowledgment

This research is supported by NSF ITR Grant ATM-0331480 and NMI ANI-0330613 and the Department of Energy Office of Science DE-FC02-01ER25492.

**Trademark, service mark, or registered trademark of Sun Microsystems, Inc., or The Open Group in the United States, other countries, or both.

References

1. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, "Web Services Description Language (WSDL)," Version 1.1, March 15, 2000; see *http://www.w3.org/TR/wsdl*.
2. D. Gannon, R. Ananthakrishnan, S. Krishnan, M. Govindaraju, L. Ramakrishnan, and A. Slominski, "Grid Web Services and Application Factories," *Grid Computing: Making the Global Infrastructure a Reality*, F. Berman, G. Fox, and T. Hey, Eds., John Wiley & Sons, Hoboken, NJ, 2003.
3. I. Foster, H. Kishimoto, A. Savva, D. Berry, A. Djaoui, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam, J. Treadwell, and J. Von Reich, "The Open Grid Service Architecture," Version 1.0, January 2005; see *www.gridforum.org/documents/GFD.30.pdf*.
4. I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke, "A Security Architecture for Computational Grids," *Proceedings of the 5th ACM Conference on Computers and Communications Security*, 1998, pp. 83–92.
5. The Globus Project; see *http://www.globus.org/toolkit/*.
6. Open Middleware Infrastructure Institute; see *http://www.omii.ac.uk*.
7. T. Oinn, M. Greenwood, M. Addis, J. Ferris, K. Glover, C. Goble, D. Hull, D. Marvin, P. Li, P. Lord, M. R. Pocock, M. Senger, A. Wipat, and C. Wroe, "Taverna: Lessons in Creating a Workflow Environment for the Life Sciences," *Concurrency & Computation: Pract. & Exper.*, Special Issue: Scientific Workflows (to be published 2006).
8. D. Churches, G. Gombas, A. Harrison, J. Maassen, C. Robinson, M. Shields, I. Taylor, and I. Wang, "Programming Scientific and Distributed Workflow with Triana Services," *Concurrency & Computation: Pract. & Exper.*, Special Issue: Scientific Workflows (to be published 2006).
9. B. Ludaescher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger-Frank, M. Jones, E. Lee, J. Tao, and Y. Zhao, "Scientific Workflow Management and the Kepler System," *Concurrency & Computation: Pract. & Exper.*, Special Issue: Scientific Workflows (to be published 2006).
10. M. Senger, "Soaplab: SOAP-Based Analysis Web Service," February 2005; see *http://www.ebi.ac.uk/soaplab*.
11. M. Senger, P. Rice, and T. Oinn, "Soaplab—A Unified Sesame Door to Analysis Tools," *Proceedings of the UK e-Science All Hands Meeting*, 2003; see *http://www.nesc.ac.uk/events/ahm2003/AHMCD/pdf/115.pdf*.
12. Axis, The Apache Software Foundation, April 10, 2005; see *http://ws.apache.org/axis*.
13. CORBA/IIOP Specification, Object Management Group, Inc., March 4, 2001; see *http://www.omg.org/technology/documents/formal/corba_iiop.htm*.
14. M. Senger, "Gowlab: Web Pages as Web Services," March 2005; see *http://www.ebi.ac.uk/soaplab/Gowlab*.
15. GridLab Products and Technologies; see *http://www.gridlab.org/about.html*.
16. Grid(Lab) Grid Application Toolkit; see *http://www.gridlab.org/WorkPackages/wp-1*.
17. SeqHound; see *http://www.blueprint.org/seqhound*.
18. BioMOBY; see *http://biomoby.org*.
19. Kyoto Encyclopedia of Genes and Genomes (KEGG); see *http://www.genome.jp/kegg/*.
20. The GridFTP Protocol and Software; see *http://www-fp.globus.org/datagrid/gridftp.html*.
21. JSR-000168 Portlet Specification (Final Release); see *http://www.jcp.org/aboutJava/communityprocess/final/jsr168*.
22. Open Grid Computing Environment, The Open Grid Computing Environments Collaboratory; see *http://www.ogce.org*.
23. I. Foster and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit," *Intl. J. Supercomputer Appl. & High Performance Computing* **11**, No. 2, 115–128 (1997).
24. GT3 GRAM Architecture; see *http://www-unix.globus.org/ogsa/docs/alpha/gram/gt3_ gram_overview.htm*.
25. A. Kropp, C. Leue, R. Thompson, C. Braun, J. Broberg, M. Cassidy, M. Freedman, T. N. Jones, T. Schaeck, and G. Tayar, "Web Services for Remote Portlets Specification," Version 1.0; see *http://www.oasis-open.org/committees/download.php/3343/oasis-200304-wsrp-specification-1.0.pdf*.
26. S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic Local Alignment Search Tool," *J. Molec. Biol.* **215**, No. 3, 403–410 (October 1990).
27. S. Graham, P. Niblett, D. Chappell, A. Lewis, N. Nagaratnam, J. Parikh, S. Patil, S. Samdarshi, I. Sedukhin, D. Snelling, S. Tuecke, W. Vambenepe, and B. Weihl, "Web Services Base Notification (WS-Base Notification)," Version 1.0; see *ftp://www6.software.ibm.com/software/developer/library/ws-notification/WS-BaseN.pdf*.

28. D. Box, L. F. Cabrera, C. Critchley, F. Curbera, D. Ferguson, A. Geller, S. Graham, D. Hull, G. Kakivaya, A. Lewis, B. Lovering, M. Mihic, P. Niblett, D. Orchard, J. Saiyed, S. Samdarshi, J. Schlimmer, I. Sedukhin, J. Shewchuk, B. Smith, S. Weerawarana, and D. Wortendyke, "Web Services Eventing (WS-Eventing)," August 2004; see *ftp://www6.software.ibm.com/software/developer/library/ws-eventing/WS-Eventing.pdf*.
29. S. Graham, P. Niblett, D. Chappell, A. Lewis, N. Nagaratnam, J. Parikh, S. Patil, S. Samdarshi, I. Sedukhin, D. Snelling, S. Tuecke, W. Vambenepe, and B. Weihl, "Web Services Brokered Notification (WS-BrokeredNotification)," Version 1.0, March 5, 2004; see *ftp://www6.software.ibm.com/software/developer/library/ws-notification/WS-BrokeredN.pdf*.
30. A. Slominski, A. di Costanzo, D. Gannon, and D. Caromel, "Asynchronous Peer-to-Peer Web Services and Firewalls," *Proceedings of the 7th International Workshop on Java for Parallel and Distributed Programming*, 2005; see *http://www.extreme.indiana.edu/xgws/papers/ws_dispatcher_ipdps2005.pdf*.
31. L. Fang, D. Gannon, and F. Siebenlist, "XPOLA: An Extensible Capability-Based Authorization Infrastructure for Grids," *Proceedings of the 4th Annual PKI R&D Workshop: Multiple Paths to Trust*, 2005; see *http://middleware.internet2.edu/pki05/proceedings/fang-xpola.pdf*.
32. J. Novotny, S. Tuecke, and V. Welch, "An Online Credential Repository for the Grid: MyProxy," *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing*, 2001, pp. 104–114.
33. B. Domenico, J. Caron, E. Davis, R. Kambic, and S. Nativi, "Thematic Real-Time Environmental Distributed Data Services (THREDDS): Incorporating Interactive Analysis Tools into NSDL," *J. Digital Info.* **2**, No. 4 (2002); see *http://jodi.ecs.soton.ac.uk/Articles/v02/i04/Domenico/*.
34. T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana, "Business Process Execution Language for Web Services," Version 1.1, May 5, 2002; see *ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf*.
35. B. Plale, D. Gannon, D. Reed, S. Graves, K. Droegemeier, B. Wilhelmson, and M. Ramamurthy, "Towards Dynamically Adaptive Weather Analysis and Forecasting in LEAD," *Proceedings of the International Conference on Computational Science*, 2005, pp. 624–631.
36. M. Xue, K. K. Droegemeier, and V. Wong, "The Advanced Regional Prediction System (ARPS)—A Multi-Scale Nonhydrostatic Atmospheric Simulation and Prediction Model. Part I: Model Dynamics and Verification," *Meteorol. & Atmospher. Phys.* **75**, No. 3/4, 161–193 (2000).
37. M. Xue, D. Wang, J. Gao, K. Brewster, and K. K. Droegemeier, "The Advanced Regional Prediction System (ARPS), Storm-Scale Numerical Weather Prediction and Data Assimilation," *Meteorol. & Atmospher. Phys.* **82**, No. 3/4, 139–170 (2003).
38. J. Michalakes, J. Dudhia, D. Gill, T. Henderson, J. Klemp, W. Skamarock, and W. Wang, "The Weather Research and Forecast Model: Software Architecture and Performance," *Proceedings of the 11th ECMWF Workshop on the Use of High Performance Computing in Meteorology*, 2004; see *http://wrf-model.org/wrfadmin/docs/ecmwf_2004.pdf*.

Received July 11, 2005; accepted for publication August 8, 2005; Internet publication March 1, 2006

Gopi Kandaswamy *Computer Science Department, Indiana University, 150 S. Woodlawn Avenue, Bloomington, Indiana 47405 (gkandasw@cs.indiana.edu).* Mr. Kandaswamy is a Ph.D. student in the Computer Science Department at Indiana University, where he is currently a Research Assistant under the guidance of Professor Dennis Gannon. He received a B.E. degree from Bharathidasan University, India, and an M.S. degree from Indiana University. Mr. Kandaswamy's current research interests include generic application factories for web services and grid workflow systems.

Liang Fang *Computer Science Department, Indiana University, 150 S. Woodlawn Avenue, Bloomington, Indiana 47405 (lifang@cs.indiana.edu).* Mr. Fang is a Ph.D. student in the Computer Science Department at Indiana University, where he is a Research Assistant responsible for investigating authorization and other security solutions to the LEAD project. He received a B.E. degree in computer engineering from Nanjing University of Science and Technology, China, and an M.S. degree in computer science from Indiana University. Mr. Fang's current research interests include grid computing, web services, portals, and their security and scalability issues.

Yi Huang *Computer Science Department, Indiana University, 150 S. Woodlawn Avenue, Bloomington, Indiana 47405 (yihuan@cs.indiana.edu).* Mr. Huang is a doctoral candidate in the Computer Science Department at Indiana University, under the guidance of Dr. Dennis Gannon. He holds a B.E. degree from Beijing University of Technology, China, and an M.S. degree from Florida State University. Mr. Huang's research interests include web services, distributed messaging systems, grid computing, system integration, and analysis of message pattern and workflow.

Satoshi Shirasuna *Computer Science Department, Indiana University, 150 S. Woodlawn Avenue, Bloomington, Indiana 47405 (sshirasu@cs.indiana.edu).* Mr. Shirasuna is a Ph.D. student in the Computer Science Department at Indiana University. He works with Dr. Dennis Gannon as a Research Assistant. He received B.S. and M.S. degrees from Tokyo Institute of Technology, Japan. Mr. Shirasuna's research interests include software tools for distributed systems, and workflow and security performance for web and grid services.

Suresh Marru *Computer Science Department, Indiana University, 150 S. Woodlawn Avenue, Bloomington, Indiana 47405 (smarru@cs.indiana.edu).* Mr. Marru is a Scientific Computing Research Specialist working with Dr. Dennis Gannon on the LEAD and Teragrid Science Gateways projects. He received a B.E. degree in electrical and electronics engineering from Osmania University, India, and an M.S. degree in electrical and computer engineering from the University of Oklahoma. Mr. Marru's research interests include distributed and grid computing, portals, web services, and developing user-friendly interfaces with atmospheric applications.

Dennis Gannon *Computer Science Department, Indiana University, 150 S. Woodlawn Avenue, Bloomington, Indiana 47405 (gannon@cs.indiana.edu).* Dr. Gannon is a professor of computer science at Indiana University. He received his Ph.D. degree in computer science from the University of Illinois and his Ph.D. degree in mathematics from the University of California. He was on the faculty at Purdue University from 1980 to 1985. Dr. Gannon's research interests include software tools for high-performance distributed systems and problem-solving environments for scientific computation.