

COOPERATING SERVICES FOR DATA-DRIVEN COMPUTATIONAL EXPERIMENTATION

The Linked Environments for Atmospheric Discovery (LEAD) project seeks to provide on-demand weather forecasting. A triad of cooperating services provides the core functionality needed to execute experiments and manage the data.

Data-driven computational science is characterized by dynamic adaptation in response to external data. Applications of this type, which are often data- and I/O-intensive, run as parallel or distributed computations on high-end resources such as distributed clusters or symmetric multiprocessing machines. On-demand weather forecasting is a canonical example of a data-driven application. As the article “Service-Oriented Environments in Research and Education for Dynamically Interacting with Mesoscale Weather” (pp. XX-YY in this issue) describes, on-demand weather forecasting is the automated process of invoking a forecast model run in response to the detection of a severe weather condition. The existing framework for running forecast models has drawbacks that we must overcome to bring about dynamic on-demand forecasting. For instance, the current investigation process requires a lot of human involvement, particularly in the staging and moving

of the files required by the model and in the invocation of downstream tools to visualize and analyze model results. Although scripts exist to automate some of the process, a knowledgeable expert must properly configure them.

The Linked Environments for Atmospheric Discovery (LEAD) project addresses the limitations of current weather forecast frameworks through a new, service-oriented architecture capable of responding to unpredicted weather events and response patterns in real time. These services are intended to support the execution of multimodel simulations of weather forecasts on demand across a distributed Grid of resources while dynamically adapting resource allocation in response to the results. At the system’s heart is a suite of core services that together provide the essential functionality needed to invoke and run a complex experiment with minimal human involvement. Specifically, it lets the user define an experiment workflow, execute the experiment, and store the results. (See the “Related Work” sidebar for a discussion of other work in this area.)

In this article, we focus on three services—the MyLEAD metadata catalog service, notification service, and workflow service—that together form the core services for managing complex experimental meteorological investigations and managing the data products used in and generated during the computational experimentation. We show how

1521-9615/05/\$20.00 © 2005 IEEE
Copublished by the IEEE CS and the AIP

BETH PLALE, DENNIS GANNON, YI HUANG,
GOPI KANDASWAMY, SANGMI LEE PALLICKARA,
AND ALEKSANDER SLOMINSKI

Indiana University

Related Work in Grid Technology

A large group of researchers has developed the Grid technology that's used as a foundation for this work.^{1,2} The LEAD project's goals and requirements are similar to several other efforts with large data management components. The Grid Physics Network (GriPhyn, www.griphyn.org) project pioneered the "virtual data" concept—data that can be automatically located from caches on a Grid or derived or re-derived from workflows executing on remote resources. GriPhyn worked with other research projects, such as the Sloan Digital Sky Survey (www.sdss.org), the Laser Interferometer Gravitational Wave Observatory project (LIGO, www.ligo.caltech.edu) for detecting gravitational waves, the International Virtual Data Grid Laboratory (iVDGL, www.ivdgl.org), and the Particle Physics Data Grid (PPDG, www.ppdg.net). Many of these physics projects have formed a larger Grid collaboration called the

Open Science Grid (<http://opensciencegrid.org>).

The UK e-science program supports another important collection of projects that are closely related to our work. Notable among these are the "MyGrid: Middleware for in silico experiments in biology" project (www.mygrid.org.uk). MyGrid has pioneered an approach to combining metadata and semantic grid technology with Web service-based workflow systems that is promising for future work in this area. The Edinburgh e-science center's Open Grid Service Architecture Data Access and Integration (OGSA-DAI, www.ogsadai.org.uk) project is a foundation for our MyLEAD system.

References

1. I. Foster and C. Kesselman, eds., *The Grid 2: Blueprint for a New Computing Infrastructure*, Morgan-Kaufman, 2nd ed., 2003.
2. F. Berman, G. Fox, and T. Hey, *Grid Computing: Making the Global Infrastructure a Reality*, John Wiley & Sons, 2003.

the services work together on behalf of a user, easing the technological burden on the scientists and freeing them to focus on more of the science that compels them. User interaction with the system is through the LEAD portal. We've shown the services in demos at several conferences, and released the MyLEAD v0.3alpha in May 2005 and the LEAD portal in June 2005.

Metadata Catalog Service

Scientists have long had to manage the data products of their experiments without help from the underlying software. For example, mesoscale meteorologists use weather observations from satellites and sensors as initial conditions. Moving the data from the machine receiving it (usually a computer at the researcher's institution) to the large multiprocessor machine that will run the forecast is largely a manual process. After the forecast model completes, the scientist must manually copy the results to his or her local file system. (For further details, see the "Data in Mesoscale Meteorology" sidebar.)

In some systems, the process is automated through large and complex scripts. This process works reasonably well until, some months or years in the future, a scientist is searching for a specific file or set of results. Retrieving specific files from prior experiments is so difficult that much of data written to long-term store is never accessed again.

The personal metadata catalog¹—the cornerstone in managing scientific data—is a collection of descriptions of the key digital products used and generated during a computational experiment, including application-specific (or semantic) metadata.

For a Doppler radar observational scan, for example, the metadata could include the scan's starting time, instrument type and spatial location, or unique four-letter mnemonic.

The catalog is organized around the notion of the experiment, loosely defined as the investigation of mesoscale weather phenomena. It can involve any number of model runs extending over several days. A MyLEAD Web service manages multiple personal metadata catalogs, handling as many as 100 catalogs at each site. The service is distributed and replicated, and interacts with other services, most notably an ontology service, to provide rich query access. By automating the metadata's organization in the catalog and exposing a familiar vocabulary for querying, we allow rich query access that doesn't require users to understand the underlying data model to write queries to it.

Service Architecture

The MyLEAD architecture can be viewed as a Web service composed of distributed services. An instance of the MyLEAD service resides at each site in a grid testbed. For example, each of the five sites in the LEAD grid runs a long-lived server-side service and a client-side service, as Figure 1 shows. The MyLEAD service at a site manages the personal metadata catalogs for users local to that site. A storage repository will reside at two sites in the LEAD grid and will be used to store the files themselves. The user interacts with the MyLEAD service through the LEAD portal, which is Web browser accessible from anywhere on the Internet.

The server (Figure 1c) is a long-lived Grid ser-

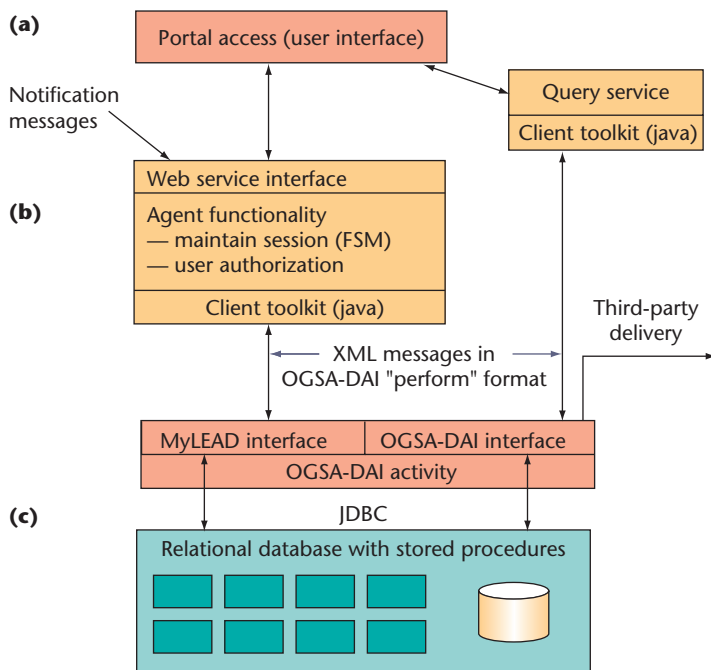


Figure 1. Service architecture and component interaction. Each site in the LEAD testbed includes a persistent server-side service and a client-side service. The MyLEAD service manages users' personal metadata catalogs.

vice built on top of a relational database. It extends the Globus Toolkit Metadata Catalog Service (MCS)² and the Open Grid Services Architecture Data Access and Integration (OGSA-DAI) grid interface layer.³ MyLEAD augments the MCS database schema with support for spatial and temporal attributes. The extensions are supported through additional methods for database access and optimizations implemented as database stored procedures. The MyLEAD agent (Figure 1b) manages stateful interactions between the user and the server.

User interaction (Figure 1a) with the service is through the LEAD portal. MyLEAD provides portlets for browsing, querying, and managing a person's personal information space. The physical data products are stored separate from the metadata catalog description in a storage repository. Query access to data objects is through a separate query service that contacts the catalog service. Results can be delivered to the user at the portal or to a service *//can't read fax//* on the user's behalf.

The storage repository could be a local file system, but storage repository solutions—the replica locator service (RLS),⁴ storage resource broker (SRB),⁵ and storage resource manager (SRM),⁶ for example—provide additional abstractions beyond

a file system API, such as a notion of a container, location-transparent data storage, and global naming. Although the MyLEAD metadata catalog could work with any of these repository tools, SRM and SRB tightly couple their own metadata catalog to their storage system, which could introduce a redundancy with costly performance implications.

MyLEAD Agent: Actively Engaged in Experiment

One of the requirements of the MyLEAD metadata catalog service is that mesoscale meteorologists be able to issue queries on application-specific terms, such as "precipitation," "vorticity," and "atmospheric pressure." With a traditional database query language like SQL, the user would have to know the names and the layout of the tables storing the metadata information. We hide that information from the user, and we can do this relatively easily by building graphical interfaces in which a user can construct a query by selecting terms that are pulled together to form a conjunctive query. We go a step beyond this, however, to give the user the illusion of hierarchical organization of the space. We do this because hierarchical organization of information is intuitive for humans. For example, telephone books have a hierarchical organization that is obviously the most useful layout of the information available in printed form.

We accomplish the structural abstraction by embedding knowledge in the agent. A MyLEAD agent works on a user's behalf during an experimental investigation. That is, it represents a user and is dedicated to a single workflow instance. A workflow instance, which we define more precisely later, can be thought of as a sophisticated script that carries out multiple steps in an experimental investigation. The agent knowledge is put to use to track the different modes, or states, of system execution (for example, model input state and model execution state). It uses this knowledge to actively organize the metadata into named buckets corresponding to that state. These named buckets can then be tied to user concepts through a dictionary maintained by an ontology service. For instance, an example concept is "model output files from second iteration." Users issue queries not only on the atmospheric terms, but also on stages in the investigation. The latter is achieved by a user selecting regions from a graphical depiction of the workflow.

The finite state machine shown in Figure 2 is used to implement the knowledge that the MyLEAD agent maintains about the experiment's current state. Transitions between states occur when status events arrive from the notification ser-

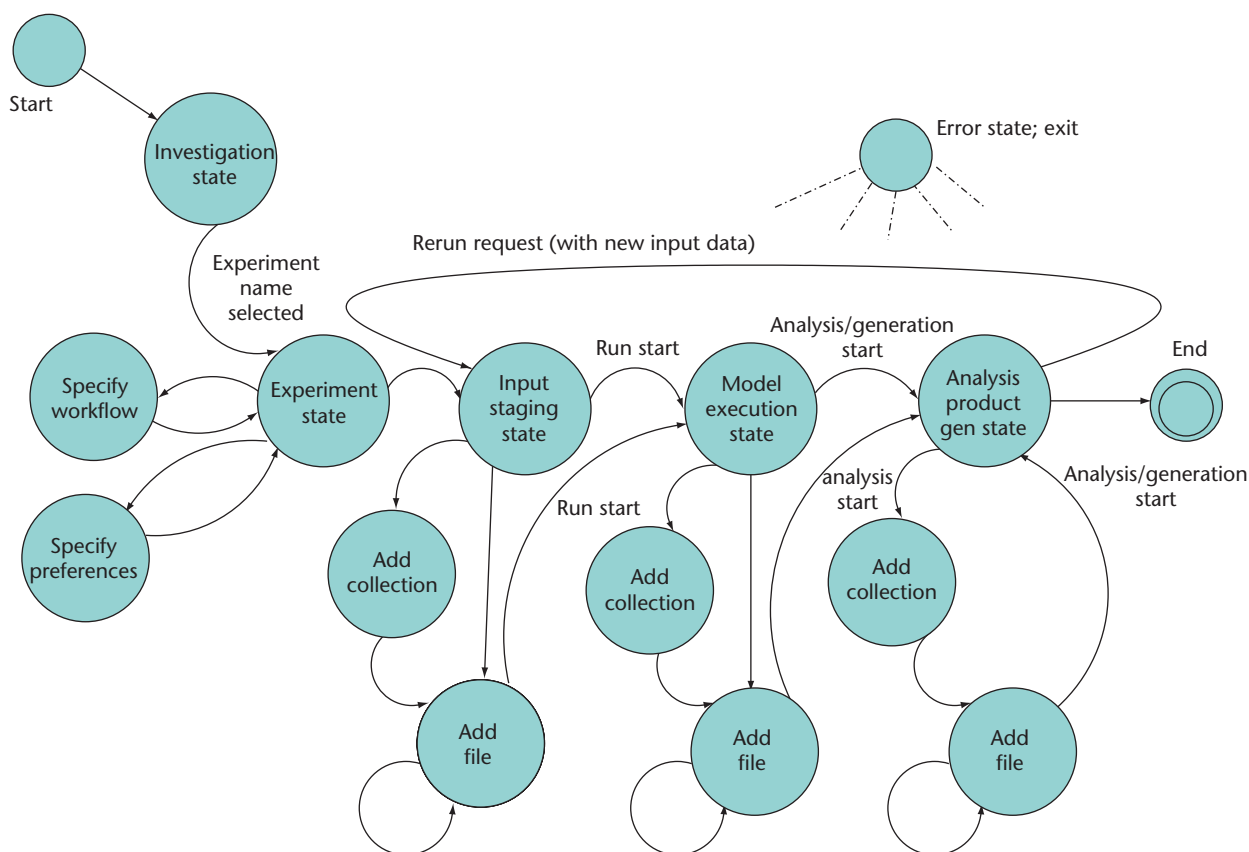


Figure 2. Finite state machine showing an experiment's current state. The MyLEAD agent encodes control flow into a finite state machine and transitions between states based on notification messages from the executing experiment.

vice or a user triggers a transition through an action at his or her keyboard. For example, a user specifies the investigation in which they wish to work. Entering the name of a new or existing experiment, or execution of a workflow, causes the portal to send a status event to the MyLEAD agent, upon which the latter transitions from the "investigation" state to the "experiment" state.

A scientist creates a computational experiment by connecting application services together graphically to form a workflow. When the workflow script is ready to be executed, the workflow engine (described later) generates an event that causes the MyLEAD agent state machine to transition to "input staging state." In this state, the agent creates a collection if none exists for the experiment run. Any files received while the agent is in the input staging state will be stored to this collection. When the agent receives an event notifying it of a transition to model execution, it transitions to the "model execution state," and creates another collection if none exists at this state.

The analysis/generation state corresponds to an

experiment's postprocessing stage, which is captured by a loop from the analysis state through file creation and can be repeated any number of times. As noted in the sidebar, analysis can trigger additional model execution rounds. The agent could reach the error state from any state in the diagram. The actual finite state machine is considerably more complex than depicted here.

Scientists can use knowledge of the experiment process to track a logical file's provenance in the catalog. Knowing the steps that went into generating a file can help scientists pinpoint sources of errors in the experimental run in case they find an anomaly in the file. Such trace information also helps peers in the scientific community understand and verify the experiment when its results are published. Scientists can correlate further notifications from the workflow engine containing a step's runtime parameters with the arrival of intermediate logical files to determine the files' complete lineage. MyLEAD can store this provenance as part of the file's metadata. In a sense, this would allow the entire experiment run to be visu-

Data in Mesoscale Meteorology

In mesoscale meteorology, researchers investigate weather phenomena such as flash floods and tornadoes using data from widespread sensors and instruments that continuously measure atmospheric conditions. The meteorologists involved in the Linked Environments for Atmospheric Discovery (LEAD) project¹ are interested in on-demand weather forecasting—that is, forecast model runs that are triggered by threatening weather conditions. These runs, which can draw on data from hundreds of observational sources, generate what meteorologists refer to as *ensemble runs*—a collection of forecast runs executing concurrently. Important technology and science factors are converging to make significant advancements in forecasting possible.

Collaborative Adaptive Sensing of the Atmosphere (CASA),² a LEAD sister project, is developing small-scale regional Doppler radars suitable for mounting on cell phone towers. These radars have a 30-kilometer radius with far higher resolution and frequency than the longer-range WSR-88D Doppler radar. Additionally, large-scale computational grids, such as Teragrid (www.teragrid.org), are sufficiently mature in their support infrastructure that harnessing numerous compute resources on demand for application processing is now feasible.

This scale of computational resource is none too late in coming as forecast models' demand increasingly large amounts of computational resources. Running a single 27-km resolution, 84-hour forecast over the US (a CONUS forecast) consumes 60 processors (30 nodes) of a

dual-processor Pentium IV Xeon 2.0-GHz cluster and takes six hours to complete. An ensemble requires significantly more resources.

For instance, at 0600hr, a research meteorologist kicks off a 2-km resolution, 12-hour regional forecast over the state of Oklahoma. As an ensemble run, 20 copies of the model are run concurrently, each with slightly different physics (or *equal*). The run completes by 0800hr, delivering 20 binary files containing temperature, wind, and microphysics. The files undergo statistical analysis to identify regions of uncertainty corresponding to regions across the state that have high levels of disagreement across the ensemble versions. Such regions can occur when too little data on the region is available. Figure A depicts the control flow.

To reduce uncertainty, meteorologists want to gather more information about the identified regions. An appropriate outcome of the statistical analysis, then, is to focus the regional Doppler radars onto the regions of uncertainty to gather additional data, as the directed line looping back to the "NetRad radar ingest" phase (Figure A) illustrates.

Suppose the radars then collect data from 0900hr to 1100hr. The application assimilation component converts and assimilates this newly collected data into the 3D input data grid (third and fourth boxes in Figure A), and kicks off another forecast at 1100hr. This time the system generates a six-hour forecast because it needs a forecast extending only through the late afternoon hours when most severe mesoscale weather occurs. The second run finishes at 1300hr and the system analyzes the ensemble results again.

ally reconstructed once it finishes, and thus let scientists drill down into sections of interest for further study.

This article focuses on the cooperation between services that occurs under the cover of the experiment to track changes, monitor progress, store results, move files, and a host of other activities that relieve the scientist of the more mundane technology-based tasks that consume so large a part of a scientist's time during a computational investigation. MyLEAD functionality goes beyond what is described in this article, however. It supports versioning of experiments. MyLEAD creates a snapshot of an experiment upon user request, and archives it in read-only form. The archived version is stripped of content that is no longer needed or can be obtained elsewhere—status messages and publicly available input files, for example. MyLEAD also lets the meteorologist or teacher share his or her files or file collections with a small group or publish products to the broader virtual organization-wide resource catalog.

Web Services-Based Notification System

As noted earlier, the MyLEAD agent is based on Web service technology. A Web service is a set of network addressable endpoints operating on XML-encoded messages. Developers use the Web Service Definition Language (WSDL)⁷ to specify the types of messages a service can receive and responses it can make. A service-oriented architecture is based on the concept of building applications by composing and orchestrating interactions between Web services. A workflow in this context is a specific template for an application based on the composition of a set of services.

Numerous services comprise the LEAD system. Two important classes of services at the application level are

- *application components*, which are services that can execute instances of specific scientific applications on compute and data resources; and
- *notification services*, which let the application

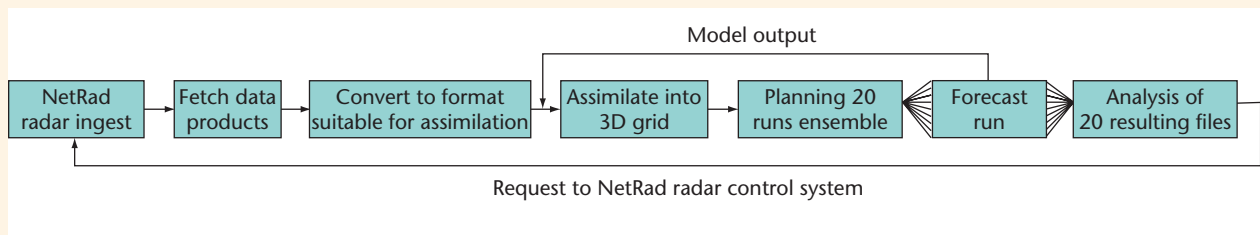


Figure A. Control flow of a mesoscale meteorology ensemble run.

This time the uncertainty is reduced, giving the meteorologist a sufficiently high degree of trust in the forecast.

This example exposes a number of challenges in provisioning for adaptive forecasts. In the flow graph in Figure A, the underlying infrastructure must fetch data products used in the forecast from where they reside. This might be the scientists' local file system or a storage repository. The Advanced Regional Prediction System (ARPS) Data Assimilation System (ADAS)³ ("Assimilate into 3D grid" in Figure A) assumes that data products are located in a named directory on the local file system. The system must organize and store products generated from the ensemble run to persistent store. It must also store the analysis results, input condition, status messages, workflow script guiding execution, and so on.

In this example, the analysis results of the prior iteration trigger subsequent loops through the graph. More pointedly, a second loop iteration is initiated based on meteorological analysis, triggering a directive to the local radars in the regions of the forecast area that have uncertainty.

A meteorology experiment's data flow contains loops, parallelism, and dynamic decisions based on application-specific criteria that can't be predicted. Using information about the data to organize metadata records is a big step toward making storage and retrieval more intuitive.

References

1. K.K. Droegeleier et al., "Linked Environments for Atmospheric Discovery (LEAD): A Cyberinfrastructure for Mesoscale Meteorology Research and Education," *Proc. 20th Conf. Interactive Information Processing Systems for Meteorology, Oceanography, and Hydrology*, 2004, http://ams.confex.com/ams/84Annual/techprogram/paper_69563.htm.
2. J. Brotzge et al., "Distributed Collaborative Adaptive Sensing for Hazardous Weather Detection, Tracking, and Predicting," M. Bubak et al., eds., *Proc. 4th Int'l Conf. Computational Science (ICCS 2004)*, LNCS 3038, Springer-Verlag, 2004, pp. 670–677.
3. K.A. Brewster, "Phase-Correcting Data Assimilation and Application to Storm-Scale Numerical Weather Prediction, Part I: Method Description and Simulation Testing," *Monthly Weather Rev.*, vol. 131, no. 3, 2003, pp. 480–492.

components publish application-specific events to the MyLEAD agent and the workflow engines.

In addition, at the system level, the MyLEAD agent itself is a service that interacts with the user at the portal and subscribes to notifications concerning the current state of the user's experimental workflow.

Publish-Subscribe Architecture

A central feature of event-driven distributed systems is a mechanism that lets one part of the system broadcast a message so that every other part of the system can learn about it. These messages, known as *events* or *notifications*, convey information such as system status, error messages, or metadata about newly generated file objects. In the world of Web services, there are two very similar standards for describing how to deliver and receive notifications: WS-Eventing⁸ and WS-BaseNotification.⁹ Both are based on a publish-subscribe architecture.

Our system uses the WS-Eventing standard,

which consists of four services:

- *Event sinks* consume events delivered to them by event sources according to subscription requests delivered to sources on behalf of the consumer.
- *Event sources* accept subscriptions for event delivery. A subscription is a message stating that a specified event sink wants to receive events from the specified source subject to the specified filter. In WS-Eventing, filters can be XPath expressions or any other Boolean expression in some custom language specific to the event source. In our case, the filter is just a topic string. If the event contains a topic string in its message header, we consider it a match for any subscriber interested in that topic.
- *Subscribers* create subscriptions and send them to event sources. (The subscriber needn't be the event sink that will receive the event stream generated by the subscription.)
- *Subscription managers* help manage services

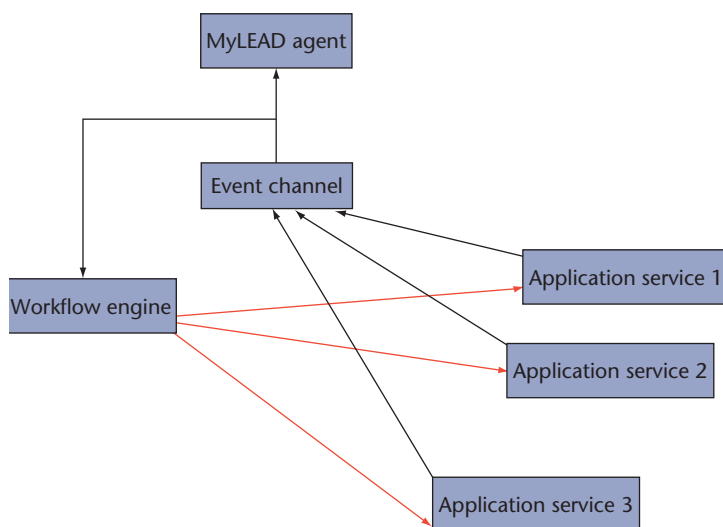


Figure 3. Event channel. The event channel is the focal point for WS-Eventing. The MyLEAD agent and workflow subscribe to messages associated with an experiment. The workflow system runs the workflow engine and application services and publishes state and metadata information, which is saved in MyLEAD. The workflow engine uses this state information to guide specific workflow instances.

once they're created. For example, to renew a subscription, check its status, or unsubscribe, a user contacts the subscription manager for the source. A subscription's response message provides the subscription manager's identity.

In the LEAD system, a central event source known as the *event channel* handles all event subscriptions. As Figure 3 illustrates, two primary event sinks exist: the MyLEAD agent and the workflow engine (the latter executes the individual workflow instances). Although the event channel is the system-wide event source, the real event sources are the workflow engine and the application services. As described earlier, MyLEAD organizes metadata around the notions of experiments, collections, and users. To associate the events generated by a specific workflow execution to the correct context in MyLEAD, the event topic string consists of the user-distinguished name and experiment name. That is, each event is uniquely identified by a <user name/Project name/Experiment name> tuple.

The event channel is the focal point of our WS-Eventing implementation. The MyLEAD agent and the executing workflow instance each subscribe to messages associated with an experiment. To run both the workflow engine and the application services, the executing workflow instance publishes

state and metadata information, which MyLEAD saves in the experiment's record. The workflow engine uses this state information to guide specific workflow instances.

Wrapping Applications as Services

Most scientific applications are conventional Fortran applications that are configured by a parameter file or command-line arguments. They aren't Web services, nor can they publish or subscribe to notifications. Consequently, to integrate them into the service architecture, we embed them in a wrapper service. We've developed an application factory service to aid in this task.

After an application has been deployed on a host, we write a simple XML service description document describing the

- path to a script to launch the application,
- parameters used to invoke the application, and
- information about notifications that are published by the application script.

Given this file, the application factory can automatically generate a Web service with which to invoke the application. This service can be invoked from a workflow or directly by a user from a Grid Web portal.¹⁰ All that's needed is authorization to run the service, which is based on Web service security protocols and values for the application parameters. The application service can run multiple instances of the application concurrently and publish the events associated with its execution to the event channel.

Among the parameters common to most applications are the names of files used as input to the applications and the Grid location for storing output. In a Grid environment, these files and directories can be remote, so we usually use URLs to describe them. The launch script moves the files to local directories where the Fortran application can find them and then monitors the application. When the application is complete, the launch script moves the output to the desired location. It also publishes event notifications about the progress, completion, and possible errors in the application execution.

We use two types of scripts to wrap the applications, both of which can send events to the event channel. One script type is an extension to the Apache Ant build tool Open Grid Runtime Engine (OGRE) under development at the US National Center for Supercomputing Applications. The extensions include the ability to move files around the Grid using the gridftp and other protocols. For example, we can use the script in Figure 4a to move a file from a source

URL to a local temporary file named "input.data" and then publish an event.

The Web service generated by the factory takes the parameter "input.source.url" from a Web service parameter when a client invokes the service. Another parameter of the Web service interface for this application is "wse.topic," the MyLEAD experiment context name for this execution.

The second type of application script is based on Jython, the Java implementation of the Python scripting language. We include this approach because Python is well received by scientific programmers who do application scripting. Jython scripts can also move files around a Grid and publish and subscribe to events. Figure 4b, for example, shows the Jython script we would use to script the file movement and event publication in the OGRE example in Figure 4a. In this case, we preinitialize the variable "context" with the MyLEAD experiment context name.

A complete experiment can require orchestrating a dozen or more of these application services, such as data decoders and data transformers, various simulation programs, data miners, and animation renderers. This orchestration is encoded in the workflow system.

Web services generated by the Application Factory are stateless. One supplies the Web service with parameters to run the application and the service executes the application. However, in some cases a user might want to interact with the application to "steer" it interactively. When an application can interact with the user through a graphical user interface, it loses this interactivity when the workflow system launches the service. Although a user could interact with a running workflow instance, integrating the original application interface into the workflow-user interaction is difficult. Discovering good protocols for integrating application interfaces into workflow executions is an interesting area of future research.

Event-Driven Workflow System

Traditional workflow systems operate by executing a fixed schedule of tasks based on a directed acyclic dependency graph. When a task completes, another

```
<uricopy from="{input.source.url}" to="/tmp/input.data"/>
<publish>
  <event message="copy to /tmp/input.data complete">
    <property name="status" value="INFORMATION"/>
    <property name="topic" value="{wse.topic}"/>
    <property name="source" value="Decoder"/>
  </event>
</publish>
(a)

sourceURL = argList[1]
cp = UriCopy()
cp.setFrom(sourceURL)
cp.setTo("/tmp/input.data")
cp.execute() notifier.sendEvent(context, "INFORMATION," "copy
to /tmp/input.data complete")
(b)
```

Figure 4. Script for wrapping applications. We use two types of script to move a file from a source URL to a local temporary file and publishing the event: (a) OGRE and (b) Jython.

set of tasks begins operating. LEAD uses two main forms of workflows. The first is based on the OGRE and Jython scripts described previously. These scripts are designed to manage workflows that are relatively short in duration (that is, not much longer than the sum of the execution times of the tasks they manage) and aren't responsible for responding to dynamic changes in their execution environment.

The more interesting form of workflow responds to external events such as resource availability and dynamically changing workloads caused by requirement changes. This adaptive, dynamic workflow is the ultimate goal of the LEAD project. For example, in LEAD, changes in the weather can determine a workflow. A data mining agent might watch a data source for special weather patterns. When it detects such an event, the agent publishes a notification that might reawaken a dormant workflow. The better-than-real-time requirements of weather predicting are another source of dynamic behavior. To meet computation deadlines, the workflow must be agile in its resource use. If tasks in the workflow require more computing, the workflow must be able to discover the appropriate computing resources and negotiate their use. This requires a set of services that monitors the entire computational Grid of available resources.

Many approaches to workflow in Grid applications exist. To achieve a more adaptive workflow capability, we use a slightly modified version of the standard Business Process Execution Language


```

<pick>
  <onMessage partnerLink="EventChannel" portType="Listener" operation="Notify" variable="notification" >
    - inspect the notification variable to see what type of event
    - this is and respond accordingly.
  </onMessage>
  <onMessage partnerLink="UserServices" portType="User-Interface" operation="kill">
    - invoke tasks which will kill off pending and running tasks
  </onMessage>
</pick>

```

Figure 5. Wait message script. We use the Business Process Execution Language to tell the workflow to wait for an event to be delivered.

(BPEL),¹¹ which was designed to orchestrate complex interactions in a service-oriented architecture.

A LEAD BPEL workflow's life cycle is simple. First, the workflow instance notifies MyLEAD that a new experiment has begun. Next, it requests available services—for example, it might request a resolver service to locate the data streams of current weather conditions. Rather than using the “please-do-this-while-I-wait-for-your-conclusion” remote procedure call mechanism, we use the more modern “literal document request” model. In this model, the workflow sends a request to an application service (for example, “Please execute the standard forecast simulation code WRF [weather research and forecast] with these parameters”). Rather than wait for the WRF engine to send the computation's results, the workflow only receives a “request acknowledged” response from the WRF service. To proceed to the next task, however, the workflow might need to wait for the results.

The BPEL basic primitives make expressing this type of interaction simple. Suppose you want a workflow that's waiting for two possible messages. One message is a notification from an agent that it has discovered a serious weather event; the other is from a user asking to change the parameters associated with a reaction to external events or suspend operations. Figure 5 shows how we encode this type of message waiting in BPEL; the language has many other simple XML tags for describing workflow sequencing, exception handling, and deployment.

In its form and structure, a BPEL workflow is not unlike a computer program that's driven by a graphical human interface: both are designed to re-

spond to events. In the case of a graphical interface, the program responds to mouse events. In the case of a BPEL program, the responses are to the workflow's partner services—that is, those with which it interacts. The language has standard program control structures like iteration and conditionals and a rich facility for exception handling.


A persistent workflow engine that saves each workflow instance's state in a database performs the actual workflow execution. After the workflow is enacted, each task is a request to a service to complete some operation. These services are the application and data Web services that eventually respond to the workflow engine with a message saying that the task has been completed. Although the workflow instance is a service, it becomes a virtual service residing in the database until the workflow receives a response. This can take minutes or hours.

The workflow can respond to different scenarios. If so designed, the workflow can wait for months to respond to a specific notification, or series of notifications, signaling that a particular configuration of weather conditions has occurred.

The workflow architecture's most fundamental limitation is that it can only execute workflows that represent programmed responses to anticipated weather scenarios. Our eventual goal is to build a truly adaptive workflow system containing various response scenario patterns that can be composed in many ways using both automated and human-assisted means.

As computational science grows, the task of managing the data products involved in scientific experimentation grows as well. Tools to manage the data must develop alongside the increasingly resource-capable computational environment. The risk in ignoring this important problem is that we severely disable the usability of the complex computational capability we are providing.

Although we've released early versions of the services to the community, some of the functionality that's more complex to implement or of a more exploratory nature is still ongoing. In MyLEAD, for instance, research in providing support for publish-

ing and sharing data products is ongoing. Atmospheric researchers will be reluctant to store data products in a new repository until they're convinced that it will preserve the privacy of their work. We're exploring a multifaceted approach to building trust that includes visual GUIs, a well-articulated security policy, and enforceable guarantees. Other areas include versioning experiments through time and improving service reliability and availability. We'll continue to work on the workflow component to improve its ability to react to unanticipated events and unplanned sequences of activities. 

Acknowledgments

We thank the LEAD team for their contributions. In particular, Jerry Brotzge, Rich Clark, Kelvin Droegemeier, Dan Weber, Bob Wilhelmson, and Sepi Yalda have been immeasurably patient in articulating the needs of meteorology researchers and educators. Jay Alameda, Doug Lindholm, Rahul Ramachandran, Lavanya Ramakrishnan, and Anne Wilson have worked tirelessly with us to define, identify, and integrate the disparate pieces of the LEAD architecture. Scott Jensen and Yiming Sun have been instrumental in the design and coding of the MyLEAD service. The US National Science Foundation funds LEAD under the following cooperative agreements: ATM-0331594 (Oklahoma), ATM-0331591 (Colorado State), ATM-0331574 (Millersville), ATM-0331480 (Indiana), ATM-0331579 (Alabama at Huntsville), ATM03-31586 (Howard), ATM-0331587 (UCAR), and ATM-0331578 (Illinois at Urbana-Champaign).

References

1. B. Plale et al., "Active Management of Scientific Data, *IEEE Internet Computing*, special issue on Internet access to scientific data, vol. 9, no. 1, 2005, pp. 27-34.
2. G. Singh et al., "A Metadata Catalog Service for Data Intensive Applications," *Proc. ACM/IEEE Supercomputing 2003 (SC 2003)*, IEEE CS Press, 2003, pp. 33-49.
3. M. Antonioletti et al., "Design and Implementation of Grid Database Services in Ogsa-dai," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 2-4, 2005, pp. 357-376.
4. S. Vazhkudai, S. Tuecke, and I. Foster, "Replica Selection in the Globus Data Grid," *Proc. IEEE/ACM Int'l Conf. Cluster Computing and the Grid (CCGRID)*, IEEE CS Press, 2001, pp. 106-113.
5. A. Rajasekar, M. Wan, and R. Moore, "Mysrb and Srb—Components of a Data Grid," *Proc. 11th IEEE High-Performance Distributed Computing (HPDC)*, IEEE CS Press, 2002, pp. 301-310.
6. A. Shoshani, A. Sim, and J. Gu, "Storage Resource Managers: Middleware Components for Grid Computing," *Proc. 19th IEEE Symp. Mass Storage Systems*, IEEE CS Press, 2002.
7. E. Christensen et al., "Web Services Description Language (WSDL) 1.1," 2001; www.w3c.org/TR/wsdl.
8. D. Box et al., *Web Services Eventing (WS-Eventing)*, <http://ftpna2.bea.com/pub/downloads/WS-Eventing.pdf>, Aug. 2004.
9. S. Graham et al., *Web Services Base Notification (WS-BaseNotification)*, 1.0, <http://devresource.hp.com/drc/specifications/wsrf/WS-BaseNotification-1-0.pdf>, June 2004.
10. D. Gannon et al., "Building Grid Portal Applications from a Web-Service Component Architecture," *Proc. IEEE*, vol. 93, no. 3, 2005, pp. 551-563.
11. T. Andrews et al., *Business Process Execution Language for Web Services*, Version 1.1, <ftp://www6.software.ibm.com/software/developer/library/ws-bpel11.pdf>, March, 2003.

Beth Plale is an assistant professor in the Computer Science Department at Indiana University. Her research interests include data management, grid computing, distributed systems, streaming data, and middleware. Plale has a PhD from the State University of New York Binghamton and completed postdoctoral studies at Georgia Institute of Technology. She is a member of the IEEE and the ACM. Contact her at plale@cs.indiana.edu.

Dennis Gannon is a professor in the Computer Science Department at Indiana University. His research interests include distributed systems, Grid computing, and building programming environments for scientific applications. Gannon has a PhD in computer science from the University of Illinois and a PhD in mathematics from the University of California, Davis. Contact him at gannon@cs.indiana.edu.

Yi Huang is a PhD student in the Computer Science Department at Indiana University. His research interests include distributed computing and programming systems. He is a member of the ACM. Contact him at yihuan@cs.indiana.edu.

Gopi Kandaswamy is a PhD student in the Computer Science Department at Indiana University. His research interests include distributed systems, Grid computing, and Web services. He is a member of the ACM. Contact him at gkandasw@cs.indiana.edu.

Sangmi Lee Pallickara is a postdoctoral researcher in the Computer Science Department at Indiana University. Her research interests include Grid computing, mobile computing, and security. Pallickara has a PhD in computer science from Florida State University. Contact her at leesangm@cs.indiana.edu.

Aleksander Slominski is a PhD student in the Computer Science Department at Indiana University. His research interests include Web services, Grid computing, and workflow systems. He is a member of the ACM. Contact him at aslom@cs.indiana.edu.