A Mechanism for Creating Scientific Application Services On-demand from Workflows

Gopi Kandaswamy Dennis Gannon
Department of Computer Science, Indiana University.
215 Lindley Hall, 150 S Woodlawn Avenue, Bloomington, IN 47405-7104
{gkandasw, gannon}@cs.indiana.edu

Abstract

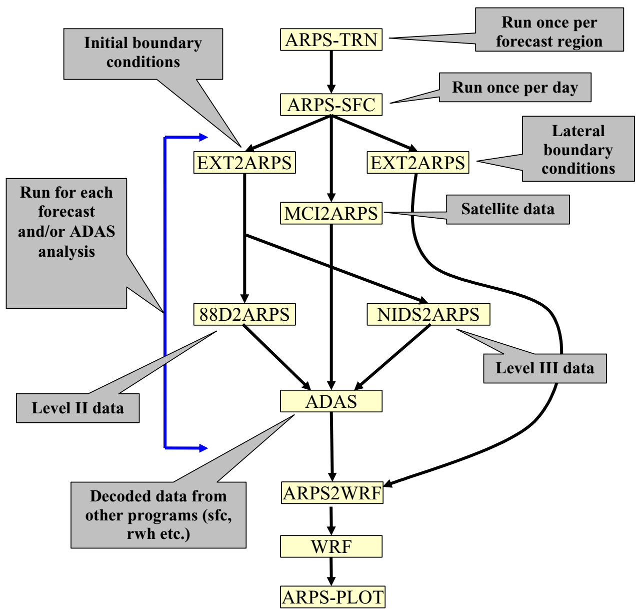
Service Oriented Computing is a new paradigm for accessing, integrating and coordinating loosely coupled software systems in a standardized way. It is increasingly being employed by large scientific collaborations to "wrap" applications as web services i.e. to create an additional "web services layer" on top of existing scientific applications. This enables scientists to easily compose, monitor and run complex workflows consisting of scientific applications that are not only developed and managed by a distributed team of application developers but also run on a distributed set of heterogeneous resources. However, one of the biggest challenges for large scientific collaborations lies in keeping all the web services persistent so that they can be accessed from scientific workflows whenever needed. In this paper we discuss the architecture and implementation of a mechanism by which we can create the web services on-demand, in the event that they are unavailable during the execution of a scientific workflow and thus obviate the need to keep them persistent.

Keywords: Web services, scientific workflows, wrapping legacy code.

1. Introduction

Web services architecture is gaining popularity in the scientific research community. It allows scientific applications to be wrapped as web services so that they can be described, discovered and consumed in a standard way. We will call such web services as application services. Application services also enable scientists to compose complex workflows from these application, execute them on a distributed set of resources on a grid and monitor their status as they run for extended periods of time. When a user invokes an operation on an application service, the application service runs the associated application, possibly on a distributed set of resources, monitors its status and returns the output results to the user. In a grid computing environment, application services often become unavailable primarily due to the unreliable nature of a grid. Sometimes even though an application service may be available, it may be unusable by a client because it does not meet the client's quality of service requirements or security policy requirements. Under such circumstances, the workflow has to be stopped and can be resumed only after the application service becomes available. During the execution of complex workflows over a period of several hours or even days, application service downtimes could result in a considerable waste of time and resources. So fault tolerance is desirable and many workflow systems like Taverna [22] , Trianna [8] and Kepler [20] can be configured to run workflows with a set of backup application services. However, these backup application services must also be running at the time of workflow execution and are not guaranteed to be available when they are actually needed. Moreover, in large scientific collaborations, owing to the large number of application services, it is unrealistic to keep all of them persistent without a huge commitment in the form of resources and support infrastructure. However, we propose that it is possible to support a small number of persistent *generic application factory* services that can create instances of any application service on-demand (just-in-time) during a workflow execution. The unique contribution of this paper is the design and implementation of our *generic application factory* (called GFac) that can create application services on-demand from workflows in a way that is completely transparent to the user and thus provides a high availability of application services without actually requiring them to be persistent.

Let us look at an example where we have used GFac to create application services on-demand. Figure 1 shows a weather forecasting workflow. It consists of 11 application services. For the sake of brevity, we will omit the application services' details. Let us assume that a scientist wants to run the workflow and so uploads it to a Workflow Execution Service (WES) and provides it with the required input data.



ARPS-TRN
Run once per forecast region
Initial boundary conditions
ARPS-SFC
Run once per day
EXT2ARPS
EXT2ARPS
Lateral boundary conditions
Run for each forecast and/or ADAS analysis
MCI2ARPS
Satellite data
88D2ARPS
NIDS2ARPS
Level III data
Level II data
ADAS
Decoded data from other programs (sfc, rwh etc.)
ARPS2WRF
WRF
ARPS-PLOT

Figure 1. A weather forecasting workflow

The WES executes the workflow by invoking its constituent application services in the order specified in the workflow, with the data provided by the scientist. Let us assume that during the execution of the workflow, the WES finds that the WRF [21] service is not available. Instead of stopping the workflow execution, the WES sends a message to GFac to create an instance of the WRF service. The WRF service is an application service and is a web service interface to the WRF application. We assume that the WRF application has been already deployed on some host. This is because neither GFac nor the application services that it creates attempt to deploy any application. GFac then creates an instance of the WRF service and returns its WSDL [7] to the WES. The WES uses WRF's WSDL to invoke it and then continues to execute the rest of the workflow. Thus, in the above example even though the WRF service was not available during the execution of the workflow, we were able to create it just-in-time using GFac, invoke it and continue executing the workflow.

The rest of the paper is organized as follows. We discuss the general architecture of GFac in section 2.1. We highlight the salient features of the application services that GFac can create in section 2.2. Details on how GFac can create application services on-demand is described in section 2.3. In section 3 we discuss the performance and scalability of GFac. In section 4, we discuss related work and we conclude in section 5.

2. The GFac

2.1 Overview of GFac

A factory service [12], is a persistent web service that knows how to create instances (possibly transient) of a particular web service. Similarly, an application factory service is a persistent web service that knows how to create instances of a particular application service. However, a *generic application factory* is a persistent web service that can create instances of **any** application service on a Grid. Our implementation of a *generic application factory* is called GFac. Before we delve into the architecture of GFac, let us briefly discuss the process of creating an application service instance and the various problems associated with it.

We can create an application service instance on a host by executing its binary on that host. But there are some problems in doing this just-in-time from scientific workflows. The first problem stems from the fact that scientific workflows are usually data intensive and compute intensive and the hosts on which the application services are to be instantiated may be determined only at run-time. For example, in weather forecasting, which involves identifying, accessing, preparing and integrating disparate and high volumes of meteorological data sets and streams, it is desirable to choose resources for running application services for data mining tasks, which are closer to the data sources. Since the host on which the application service will be created is not known before-hand, the problem is that we either need to install the application service binary on all the hosts on a Grid or download it from some repository just-in-time. Owing to the large number of application services in a scientific community, the former may be a highly involved task and depending on the size of the binary and the speed of the network, the later may take anywhere from several seconds to a few minutes. The second problem is that the application service binary is not guaranteed to run on the target host. Even for Java based web services, due to the many incompatibilities that exist between the different versions and implementations of the Java Virtual Machine, unforeseen problems can occur at run-time. We can overcome this problem by compiling the application service's source code just-in-time on the target host. But this only increases the over-head of creating the application service instance, which may be undesirable under many circumstances. The third problem lies with the basic assumption in this approach; the assumption that either the application service's source code or binary is available for us to begin with. In many situations we cannot make this assumption. This is because most scientific applications are command line applications and although the task of "wrapping" an application as an application service is not difficult for a specialist trained in web services, for most application developers it is a significantly high barrier to pass. There are several tools that can help application developers "wrap" their applications as application services, but owing to the large number of applications in a scientific community, maintaining different versions of the source code and binaries for all the application services is a difficult task.

Our GFac adopts a new approach. When it receives a request from a client to create an application service instance on some host, it instantiates a *generic service* binary that is pre-installed on that host. GFac then provides the *generic service* instance with a configuration document that "describes" the application service. We call this configuration document the ServiceMap document. The ServiceMap document is not a WSDL. It is a higher level language than WSDL for describing the "WSDL portType", the security policies and soft-state lifetime management policies of an application service. It is described in detail in [18]. It is written by the application service provider (also known as service provider or application provider in this context) and registered with a well known Registry service so that it can be retrieved by GFac for creating an instance of that application service. Using the ServiceMap document, the *generic service* instance configures itself to "become" the application service instance. The application service instance then generates its WSDL and registers it with a Registry service. GFac then returns the application service's WSDL in the response message to the client. The client can then use the application service's WSDL to invoke it directly.

Using the above approach, GFac can create any application service instance from the *generic service* binary. This approach also significantly reduces the overhead of creating an application service instance just-in-time. There are some important things to note here.

- The *generic service* is a Java based web service. Its binary has to be pre-installed on all the hosts on which we might create application services. But this has to be done just once.
- GFac is a secure factory service. It supports two security mechanisms.
  - Transport Level Security (TLS): X509 certificates are used for authentication. All authenticated users are allowed access to all operations on GFac i.e. no fine grained authorization is used to decide which user has access to what operations on GFac. We will call this "service level" authorization.
  - Message signature with authorization tokens: Authentication is using X509 certificates and authorization is using XPOLA [11], which uses SAML [2] tokens for authorization. Authorization tokens are used to decide which user has access to what operations on GFac. We call this "operation level" authorization.
- GFac instantiates the *generic service* binary on the target host using GRAM. This requires a globus gatekeeper service to be running on the target host. A service container is not required to be installed or running on the target host because the *generic service* binary has a small embedded HTTP container provided by XSUL [26].
- GFac supports both synchronous and asynchronous modes of invocation through the use of WS-Addressing [6].

2.2 Overview of the application services created by GFac

GFac can wrap any command-line application as an application service. We list below some of the salient features of the application services created by GFac. Details on the architecture of the *generic service* , its security mechanisms and how it has been used in the LEAD [10] [1] project for wrapping data decoders, data mining tools, weather simulations and graphical rendering engines for use in weather forecasting workflows, can be found in [18], [23] and [13].

- The application services are secure. They support the same security mechanisms that GFac supports viz. TLS and message signature with authorization tokens.
- The application services do not attempt to deploy any application. So the command-line application that the application service "wraps" has to be pre-deployed and must be ready-to-run on some host. The deployment of the application is usually done by the application provider.
- The deployment of the application is described in an XML document called the "ApplicationDeployment-Description" document and has to be registered with a well known Registry service so that it can be retrieved later by the application service to run the application. The "ApplicationDeploymentDescription" document apart from other details, contains the name of the host on which the application is deployed, the path to the application on that host and the environmental variables that are needed to run the application on that host.
- The application services can run their applications as batch jobs using schedulers like PBS, LSF, Condor and SGE through the use of GRAM. Schedulers like SLURM [16] are also supported by built-in adapters in the application services.

- The application services can stage the input data files before running the application and the output data files after running the application.
- The application services can send notifications about their status and the status of their applications to a well known Notification service using WS-Eventing [5] and WS-Notification [14]. Interested clients can subscribe to the Notification service to get these notifications.
- The application services automatically generate a graphical user interface in the form of a HTML page. The graphical user interface describes all the operations that the user can invoke, allows the user to choose an operation, specify its input parameters and invoke the operation on the application service.
- The application services provide soft-state lifetime management. They renew their WSDL's with a well known Registry service.
- The application services have built-in "shutdown" and "kill" operations that can be invoked to shutdown or kill the service. The shutdown operation unregisters the WSDL from the Registry service, waits for all the jobs (application instances) started by the service to finish and then stops the service. The kill operation unregisters the WSDL from the Registry service, kills all running jobs and stops the service.
- The application services support both synchronous and asynchronous modes of invocation through the use of WS-Addressing.
- Each application service instance can support upto 250 concurrent clients using synchronous request-response model for invocation and upto a 1000 concurrent clients using asynchronous request-response model for invocation.

2.3 Creating application services on-demand from workflows

As we mentioned in section 1, if an application service that is part of a workflow is not available during a workflow execution, it can be created just-in-time using GFac. In Figure 2, the Workflow Execution service (WES) wants to create an instance of an application service. So it queries a well known Registry service to obtain the WSDL for GFac. It then sends a SOAP message to GFac in step 1. The SOAP message contains the fully qualified name of the application service whose instance it wants to create. After receiving the message, GFac verifies its authenticity and ensures that the WES is an authorized user. In step 2, GFac queries the Registry service to obtain the ServiceMap document for the requested application service. If no ServiceMap document is found, an application service instance cannot be created and GFac returns a SOAP fault to the WES. If a ServiceMap document is found, it is validated by GFac. Now, GFac needs to know on which host it should create the application service instance. Ideally, GFac should be able to contact another service to get this information. Since we do not have such a service in our current system, GFac can be configured to choose any host from a grid or any host from a specified group of hosts on a grid. After determining the host on which to create the application service instance, in step 3, GFac queries the Registry service to obtain the "HostDescription" document for that host. It is an XML document that apart from other details, contains the path to the *generic service* binary on that host. In step 4, GFac instantiates the *generic service* binary on the target host using GRAM and provides it with the "fully qualified name" of the ServiceMap document. The *generic service* instance then retrieves the ServiceMap document from the Registry service and configures itself to become the application service. After the configuration is done, the application service generates its WSDL and registers it with the Registry service in step 5. GFac then queries the Registry service to obtain the WSDL for the application service in step 6. In step 7, GFac returns the WSDL for the application service in the response message to the WES. The WES can then use the application service's WSDL to invoke it in step 8.

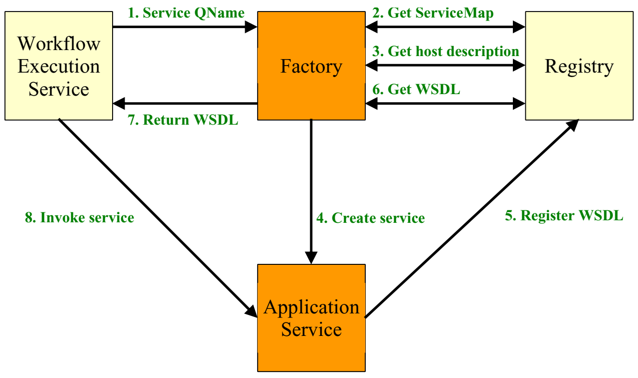


Figure 2. Just-in-time creation of application services using GFac

There are a few important things to note here. First, as mentioned in section 2.1, since the *generic service* binary is pre-installed on the host, there is no need to download it before instantiating it. This greatly reduces the overhead of creating an application service instance. Second, no web service container needs to be installed or running on the host on which the application service is created. This avoids the hassles of hot-deployment and hot-update of web services

Table 1. Performance of GFac in the three security modes

Test	Security	Time (millisecs)
Test-1	No security	2221
Test-2	TLS	2585
Test-3	Msg. sig w/ auth. tokens	2935

in a container. Third, there is an overhead involved in dynamically configuring the *generic service* instance but as we will see in section 3 this overhead is small. Fourth, the process of creating the application service just-in-time is completely transparent to the user.

3. Performance and scalability tests on GFac

For performance and scalability tests, we used a cluster of nodes for running the clients, GFac and the application services. Each node in the cluster had two 64 bit AMD Opteron processors running at 2 GHz and 8 GB of memory. We used 32 bit Sun JDK 1.4.2 for the tests.

3.1 Performance tests

For performance tests, the client and GFac were running on two different nodes in the cluster and GFac created the application services on a different set of nodes in the cluster. The client used synchronous request-response model to send a request and receive a response. Three performance tests were conducted. In Test-1, GFac was running in the unsecure mode. In Test-2, GFac was running in the TLS mode and in Test-3 GFac was running in the message signature with authorization tokens mode. In each test, the client sent 1 request to GFac to create 1 application service. The time elapsed between the moment the client started sending the request and the moment the client finished receiving the response (with the WSDL for the newly created application service instance) was measured. The three tests were repeated several times and the average time to create an application service is shown in Table 1.

We see from Table 1, that the time to create 1 application service using GFac (in unsecure mode) is just 2.221 seconds. There are a few important things that we would like to mention here. First, most application services run their applications on clusters as batch jobs. The overhead introduced by batch queues is usually high. Second, many scientific applications take several minutes if not hours or days to run. Third, once an application service is created, we can use it to the run the application how many ever times we want before shutting it down. So the overhead of 2.221 seconds for creating an application service just-in-time is quite acceptable under most circumstances.

3.2 Scalability tests

As in the performance tests, for the scalability tests, the client and GFac were running on two different nodes in the cluster and GFac created the application services on a different set of nodes in the cluster. We started 1 client with several threads that accessed GFac concurrently. We used a maximum of only 50 threads in the client because we felt that in real situations it is unrealistic for GFac to receive concurrent requests to create more than 50 different application service instances (Note: Each application service instance can support upto 250 concurrent clients using synchronous request-response model of invocation and 1000 concurrent clients using asynchronous request-response model of invocation). Each client thread sent 1 request to GFac using **synchronous request-response model**, over a **separate HTTP connection**, to create 1 application service instance. So effectively, each client thread is a separate client. The time elapsed between the moment a client started sending the request and the moment the client finished receiving the response (with the WSDL for the newly created application service instance) was measured. This is the **response time** of GFac and is measured on the client side. We mentioned in section 2.3 that GFac can be configured to create application services on a specified group of hosts on a grid. For our tests, we will call these hosts as "service nodes" i.e. the nodes in our cluster on which GFac creates application services.

Figure 3 shows the response time of GFac. There are 4 graphs; Test-1, Test-2, Test-3 and Test-4 where 4, 8, 16 and 25 service nodes were used respectively. We see from Test-1 that the average response time of GFac when only 1 client is accessing GFac is 2.1 seconds. This increases to 12 seconds when 50 concurrent clients are accessing GFac. The increase in response time is due to lack of enough resources (service nodes) to create application services. By increasing the number of service nodes, the response time can be reduced. This can be seen from Test-2, Test-3 and Test-4. In Test-4, we see that even with 50 concurrent clients, the average response time is under 4 seconds.

It is important to note that the response time of GFac shown in Figure 3 is the **total** time as seen by a client to create an application service instance (using synchronous request-response model of invocation). It includes the time taken by the following steps.

1. Client sends request to GFac to create an application service instance.

2. GFac gets the ServiceMap document for the application service from the Registry service and validates it.

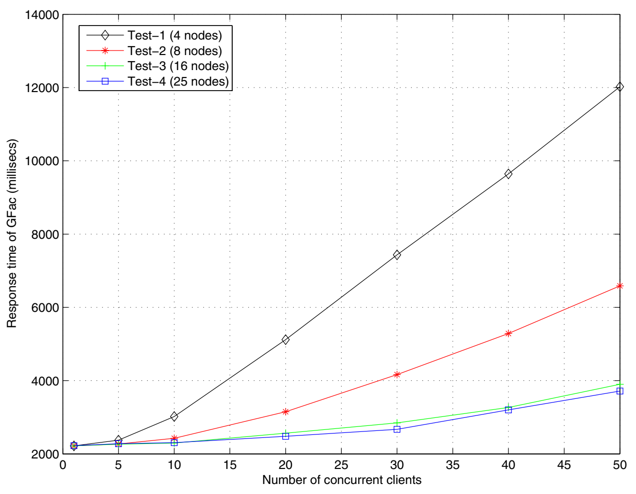


Figure 3. Response time of GFac

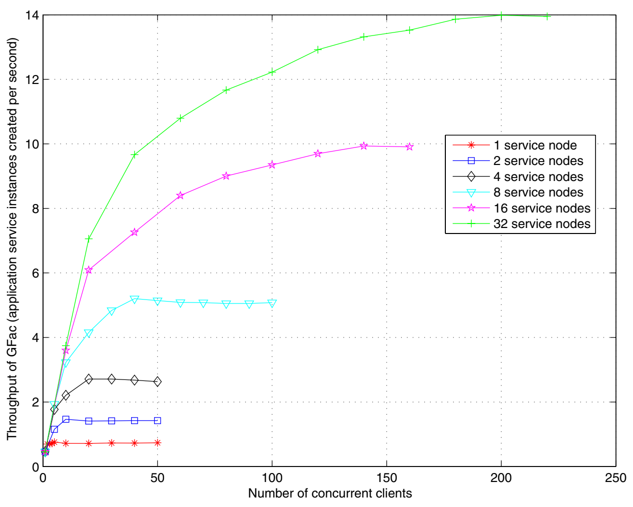


Figure 4. Throughput of GFac

3. GFac instantiates the *generic service* binary on the remote host.

4. The *generic service* instance gets the ServiceMap document for the application service from the Registry service.

5. The *generic service* instance "configures" itself using the ServiceMap document to become the application service instance.

6. Application service instance generates its WSDL.

7. Application service instance registers its WSDL with the Registry service.

8. GFac obtains application service instance's WSDL from the Registry service.

9. GFac returns the application service instance's WSDL to the client.

To see how far GFac is scalable, we measured its thoroughput and saw how it varies with an increase in the number of concurrent clients accessing it. Here again, the concurrent clients used synchronous request-response model to invoke GFac. The throughput of GFac is the number of application service instances that it can create in one second and is calculated at the service side i.e. in GFac and is shown in Figure 4. The first graph (from the bottom of the figure) shows the throughput of GFac when 1 service node is used. The second graph is the throughput when 2 service nodes are used. We see that the throughput of GFac depends on the number of service nodes used to create application services. The more the number of service nodes, the higher is the throughput. This is intuitive. But whatever be the number of service nodes, we see that the throughput of GFac increases with increase in the number of concurrent clients, reaches a maximum value and then either decreases very slowly or remains constant.

Based on the results of the performance tests we can conclude that the time taken by GFac to create an application service instance is small enough to be used just-in-time from workflows. Based on the results of the scalability tests, we can conclude that a single instance of GFac is scalable upto 200 concurrent requests (using synchronous request-response model), although we feel that in real-applications it is unrealistic for GFac to receive more than 50 concurrent requests.

4. Related Work

Over the last few years there has been some progress in designing and building generic tools that can wrap any scientific application as an application service and instantiate it on a Grid. But none of them can create an application service instance just-in-time during a workflow execution.

Although the SoapLab [25] toolkit can be used to wrap applications as application services, the process is not automated. Also, SoapLab uses Apache Axis to create Java implementation classes for the application services which incurs a significant overhead. Also, the services need to be installed in a web services container like Tomcat. Although Tomcat supports remote deployment of web services, there is some overhead in doing the remote deployment. Moreover, this assumes that a Tomcat container is present on the host on which we want to create the application service instance.

GridDeploy [15] is a toolkit that provides a grid service interface to applications. When a client request to run an application is received by the GridDeploy factory, a grid service is created within the same Grid services container as the factory. The factory then redirects the client's request to the grid service. The user then invokes the grid service to run the application. After executing the application, the factory shuts down the service and cleans up the user environment. The architecture of the GridDeploy factory is not scalable enough for most applications. This is because the GridDeploy factory can create the application service only on the local host and within the same container as that of the GridDeploy factory. This limits the number of application services that can be created dynamically and hence affects the scalability of the system.

Another system that provides a Web service interface to legacy scientific applications is the Generic Application Service (GAP) [24] which is a part of the In-Vigo [4] system. A user executes an application, by sending a request to the GAP service with the name of the application. The GAP service retrieves a description of the application from a repository and presents the user with a graphical user interface for providing the command line arguments to the application. GAP then submits the application as a job to the In-Vigo system. Although GAP serves to provide a web service interface to any application, it is important to note that GAP is not an application specific service. It can be considered to be a Generic Application service. Its WSDL is not specific to any application service and cannot be used to compose workflows.

GEMLCA [9] [17] is a system that can deploy any legacy application as an OGSI [27] service without code re-engineering. This system is similar to SoapLab in the sense that the process of actually 'wrapping' the application as a grid service is not truly automated and the grid service cannot be deployed on a remote host.

[19] describes an end-to-end system for integrating multi-scale bio-medical applications using a service oriented architecture. It uses Apache Axis [3] to wrap the applications as web services and uses Tomcat as the hosting container for these web services.

5. Conclusions

GFac can create application service instances just-in-time during workflow execution. It also overcomes the limitations of currently available tools to "wrap" command-line applications as application services as follows.

- GFac creates application service instances not by instantiating the application service binary but by instantiating a *generic service* binary and "configuring" it just-in-time. This is a novel way of "wrapping" a command-line application as an application service and is done just-in-time and in a manner that is completely transparent to the user. There is no need to download, generate or compile any application service code. It is simple and cost-effective as it eliminates the need to maintain and support the source code and binaries for a large number of application services in a scientific community.
- The total time to create an application service instance is small and is acceptable for creating most application services just-in-time during a workflow execution.
- GFac has the ability to create the application service instance remotely on any host on a grid without the requirement that the host must have a web or grid service container. This avoids the hassles associated with hot-deployment and hot-update of services in a web or grid services container.

References

[1] Linked Environments for Atmospheric Discovery. http://www.lead.ou.edu/.

[2] Security Assertion Markup Language (SAML) v1.1. http://www.oasis-open.org/specs/index.php#samlv1.1.

[3] Web Services - Axis. http://ws.apache.org/axis.

[4] S. Adabala, V. Chadha, P. Chawla, R. Figueiredo, J. Fortes, I. Krsul, A. Matsunaga, M. Tsugawa, J. Zhang, M. Zhao, L. Zhu, and X. Zhu. From virtualized resources to virtual computing grids: the In-VIGO system. *Future Generation Computer Systems*, 21(6), April 2005.

[5] 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. ftp://www6.software.ibm.com/software/developer/library/ws-eventing/WS-Eventing.pdf.

[6] D. Box, E. Christensen, F. Curbera, D. Ferguson, J. Frey, M. Hadley, C. Kaler, D. Langworthy, F. Leymann, B. Lovering, S. Lucco, S. Millet, N. Mukhi, M. Nottingham, D. Orchard, J. Shewchuk, E. Sindambiwe, T. Storey, S. Weerawarana, and S. Winkler. Web Services Addressing (WS-Addressing), August 2004. http://www.w3.org/Submission/ws-addressing/.

[7] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL), Version 1.1, March 2000. http://www.w3.org/TR/wsdl.

[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 and Computation: Pract. and Exper., Special Issue: Scientific Workflows*, 2006. To be published.

[9] T. Delaitre, A.Goyeneche, P. Kacsuk, T.Kiss, G. Terstyanszky, and S. Winter. GEMLCA: Grid Execution Management for Legacy Code Architecture Design. In *Proceedings of 30th EUROMICRO Conference*, August 2004.

[10] K. K. D. et al. Linked environments for atmospheric discovery (LEAD): A cyberinfrastructure for mesoscale meteorology research and education. In *20th Conf. on Interactive Info. Processing Systems for Meteorology, Oceanography, and Hydrology*, January 2004.

[11] L. Fang, D. Gannon, and F. Siebenlist. XPOLA: An Extensible Capability-Based Authorization Infrastructure for Grids. In *Proceedings of the 4th Annual PKI R&D Workshop: Multiple Paths to Trust*, 2005.

[12] D. Gannon, R. Ananthakrishnan, S. Krishnan, M. Govindaraju, L. Ramakrishnan, and A. Slominski. *Grid Computing: Making the Global Infrastructure a Reality*, chapter 9, Grid Web Services and Application Factories. John Wiley and Sons, 2003.

[13] D. Gannon, B. Plale, M. Christie, L. Fang, Y. Huang, S. Jensen, G. Kandaswamy, S. Marru, S. L. Pallickara, S. Shirasuna, Y. Simmhan, A. Slominski, and Y. Sun. Service Oriented Architectures for Science Gateways on Grid Systems. In B. Benatallah, F. Casati, and P. Traverso, editors, *Proceedings of International Conference on Service Oriented Computing*, pages 21–32. Springer-Verlag Berlin Heidelberg, 2005.

[14] 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. ftp://www6.software.ibm.com/software/developer/library/ws-notification/WS-BaseN.pdf.

[15] Z. Guan, V. Velusamy, and P. Bangalore. GridDeploy: A Toolkit for Deploying Applications as Grid Services. In *Proceedings of International Conference on Information Technology: Coding and Computing (ITCC'05)*, volume 2, pages 764–765, 2005.

[16] M. Jette and M. Grondona. SLURM: Simple Linux Utility for Resource Management. In *Proceedings of ClusterWorld Conference and Expo*, June 2003.

[17] P. Kacsuk, A. Goyeneche, T. Delaitre, T. Kiss, Z. Farkas, and T. Boczko. High-level Grid Application Environment to Use Legacy Codes as OGSA Grid Services. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, November 2004.

[18] G. Kandaswamy, L. Fang, Y. Huang, S. Shirasuna, S. Marru, and D. Gannon. Building web services for scientific grid applications. *IBM Journal of Research and Development*, 50(2), 2006. To be published.

[19] S. Krishnan, K. Baldridge, J. Greenberg, B. Stearn, and K. Bhatia. An End-to-End Web Services-Based Infrastructure for Biomedical Applications. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, 2005.

[20] 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 and Computation: Pract. and Exper., Special Issue: Scientific Workflows*, 2006. To be published.

[21] 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. In *Proceedings of the 11th ECMWF Workshop on the Use of High Performance Computing in Meteorology*, 2004.

[22] 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 and Computation: Pract. and Exper., Special Issue: Scientific Workflows*, 2006. To be published.

[23] B. Plale, D. Gannon, Y. Huang, G. Kandaswamy, S. Pallickara, and A. Slominski. Cooperating services for data driven computational experimentation. *Computing in science and engineering*, 7(5):34–43, 2005.

[24] V. Sanjeepan, A. Matsunaga, L. Zhu, H. Lam, and J. A. Fortes. A Service-Oriented, Scalable Approach to Grid-Enabling of Legacy Scientific Applications. In *Proceedings of 2005 International Conference on Web Services (ICWS-2005)*, pages 553–560, July 2005.

[25] M. Senger, P. Rice, and T. Oinn. Soaplab: A Unified Sesame Door to Analysis Tools. In *Proceedings of the UK e-Science All Hands Meeting*, 2003.

[26] A. Slominsky and L. Fang. WS/XSUL2: Web and XML Services Utility Library (Version 2). http://www.extreme.indiana.edu/xgws/xsul/.

[27] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maquire, T. Sandholm, D. Snelling, and P. Vanderbilt. Open Grid Services Infrastructure (OGSI) Version 1.0. http://www-unix.globus.org/toolkit/draft-ggf-ogsi-gridservice-33_2003-06-27.pdf.