Gluing Grids and Clouds together: A Service-oriented Approach

Ashiq Anjum, Richard Hill
School of Computing and Mathematics
University of Derby. Derby, UK
Email: {a.anjum, r.hill}@derby.ac.uk

Richard McClatchey
The Centre for Complex Cooperative Systems (CCCS), UWE Bristol, UK
Email: richard.mcclatchey@cern.ch

Abstract:
Scientific communities are actively developing services to exploit the capabilities of service-oriented distributed systems. This exploitation requires services to be specified and developed for a range of activities such as querying, management and scheduling of workflows, image handling, provenance capture and management, amongst other activities. Most of these services are designed and developed for a particular community of scientific users. A key issue here is that the constraints imposed by architectures, interfaces or platforms can restrict or even prohibit the free interchange of services between disparate scientific communities who have a shared need for a particular service. Using the notion of "Platform as a Service" (PaaS), we propose an architectural approach that addresses these limitations so that that users can make use of a wider range of services without being concerned about the development of cross-platform middleware, wrappers or any need for bespoke applications in order to cross the boundaries between differing communities. The proposed architecture shields heterogeneous Grid and Cloud infrastructural detail with a brokering environment, thus enabling users to concentrate on the specification of higher level services that are more relevant to the intended application.

I. INTRODUCTION

Scientific communities are increasingly building high level distributed services such as querying services, workflow and scheduling services, analysis services and provenance services. These services or applications help scientific users to analyze their data, to extract knowledge and to share it with other users. Most of these services are designed and developed for a particular computing platform (such as the glite Grid middleware [1] or the OpenStack cloud platform [2]) and for a specific community of scientific users. It is often difficult to share or re-use services from one community with another community, due to architecture, interface or platform limitations. Similarly, the services developed for one platform cannot be deployed in another computing environment due to stack mismatch and interoperability issues between the service providers. This has become particularly critical in the case of distributed infrastructures such as Grids and Clouds that host a number of high level applications to facilitate users. An application designed and developed for one Grid middleware is often challenging to run on another platform thus limiting the wide scale adoption of this technology by the wider community. This is particularly prevalent for Cloud providers, since there is a tendency to adopt proprietary standards for service implementation, thus inhibiting generalization of the service specification, with consequent cloud platform lock-in. As Grid technologies evolve onto Cloud platforms the situation worsens further since either the applications need to be redeveloped for the emerging platforms or significant changes are required to adapt the applications for the newer platforms. The potential wasted effort as well as the added investment required presents a significant barrier to the adoption of Cloud technologies.

Applications for Grids are relatively tightly coupled to the underlying platform. This presents two significant issues. Firstly, typical Grid services such as resource management, scheduling and security policy management are specific to the platform and therefore users must acquire and maintain specific expertise related to the Grid infrastructure used. Secondly, an application that itself exhibits heterogeneity before deployment to a Grid requires specific implementation that further increases the coupling of the application to a particular infrastructure.

As a result of this, any changes in platform middleware, or a desire to move from one Grid provider to another, will require a significant amount of application re-development. Consequently users have a fundamental dilemma in terms of weighing up the eventual value of the system against fundamental application re-development costs. This clearly does not enable the benefits of a service-oriented approach to be realized, whereby users should be concentrating upon their core business, rather than the engineering requirements of their underlying infrastructure.

The marketing hype surrounding Grid and Cloud Computing often promises elasticity of resources, ubiquitous access and abstraction from heterogeneity [3]. The reality is often somewhat different, and as long as infrastructure interoperability remains a challenge [4], the benefits of detail abstraction cannot be realized. For instance, if an application's needs cannot be addressed by a local Grid/Cloud infrastructure, the ability to add capability on demand by calling upon Grid or Cloud services, irrespective of where and how they are hosted, is still problematic and far from the promise of utility computing. As described earlier, service based infrastructures tend to be constrained by the suppliers of the platforms, thus servicing as a disincentive towards the utilization of more than one Grid or Cloud platform. If an application is built upon a combination of local resources and commercial resources (such as RackSpace [5] or Amazon [6]), any future or emerging business requirement to adapt or transfer to other platforms will require application re-development, which does not faithfully follow a service-based model of computing. A purer form of PaaS should enable the users to run their applications as a utility consumption, without being concerned with whoever is providing the infrastructure, what platform or protocol has been used or how to switch to a different computing infrastructure provider. The non-availability of such an environment is therefore a fundamental barrier towards the pervasive adoption of Grid and Cloud platforms.

This is far from ideal for users, who not only have preferences in using tools and technologies for their analyses, but they are resistant towards the introduction of new interfaces or attempts to customize the applications. This takes up significant time and resources in understanding the platforms and technologies which should be spent on solving the scientific or business challenge. Essentially, this trend drives users away from their core business and discourages them from exploiting Grid or Cloud resources. If the decision is made to exploit these alternative platforms, then users must invest considerable time to understand the platform, its interfaces, underlying protocols and technologies before high level applications to exploit the compute and data resources available can be written. There is no platform or service available that can enable users to abstract these details so that they only focus on the application level details with the underlying complex infrastructure details remaining hidden (unless, of course, it is important for increasing the performance of an application or for other users' needs). The problem is exacerbated by frequently changing platforms, technologies and interfaces which force users to redesign and re-implement their applications. This restricts the adoption of computing platforms made available through Grid and Cloud technologies, which are eagerly anticipated as the deployment platforms for new applications.

In this paper, we propose a platform that addresses the limitations stated above. We extend the notion of Platform as a Service (PaaS) to address platform, interface and technology limitations, to ensure that users do not need to worry about the lower level details of a platform for cross community applications. Using a service-oriented platform, a brokering environment is offered which shields the heterogeneity of distributed resources in present in Grids and Clouds. This platform (referred to as the Gluing Platform (GP) thereafter) supports applications by allowing concurrent access to a number of platforms, by shielding the underlying infrastructure details from users and applications, thus enabling users to implement higher level services. The platform provides features such as workflow enactment, execution control and monitoring, provenance and tracking, concurrency management at the application level, scalability to support a number of applications and users, and security.

The GP exposes the SAGA (Simple API for Grid Applications) [7] implementation as a distributed service using the service-oriented paradigm. Briefly, the key advantages of the GP are:

1. It provides (using SAGA) a standard way of accessing Cloud and Grid services without tying down services and applications to a particular Cloud or Grid middleware.
2. It enables access to any deployed Grid or Cloud middleware through a service-oriented brokering environment.
3. It offers a solution that enhances reusability of existing services across domains and applications.
4. It facilitates development productivity by offering a service based approach to shield users from complex Grid and Cloud specific functionality.
5. It offers an easy to use approach for enabling clients to port their applications

to Grids or Clouds (or both) without installing and maintaining too many Grid and Cloud specific libraries.

The paper proceeds as follows. Section II, discusses the GP architecture and describes the considerations that went into the design of the platform. Section III describes an exemplar case study about the usage of the platform. The services and components in the platform are then described in Section IV. These are the core components that enable the user to access the platform functionality. Section V concludes the paper and identifies opportunities for future work.

II. The Gluing Platform Architecture

A high level architecture for the Gluing Platform is shown in Figure 1. The GP sits between infrastructure and the applications and provides the required independence from the underlying platforms. A user does not need to know about the lower level APIs or protocols, components and services are provided to enable access to the underlying compute and resources in a Grid or Cloud. The services in the platform include authentication and authorization services, scheduling and resource management services, monitoring and provenance services, data access and browsing services and enactment and execution services. To interact with heterogeneous middleware, an adapter based approach has been utilized, that implements middleware for each supported Grid or Cloud in the GP. The GP enables applications to be platform agnostic by exposing suitable interfaces that can be accessed to run an application, to browse data, to monitor the execution details or to retrieve the provenance history. The API details will be described in the following sections.

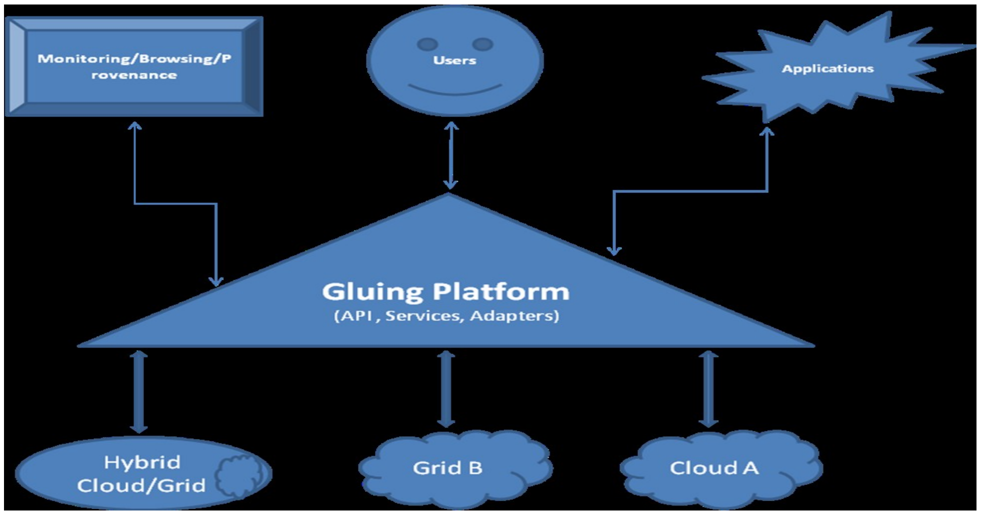

Figure 1: Gluing Platform Architecture

The GP exposes the SAGA (Simple API for Grid Applications) implementation as a web service to address the objectives described previously. SAGA is an open standard that is defined and maintained by the Open Grid Forum [8] (OGF), which describes a high-level interface for effortless programming of Grid/Cloud applications. SAGA is not designed for middleware developers; rather it provides APIs for developing applications using different Grid or Cloud middleware. The SAGA API permits a job to be executed on a particular middleware by the runtime loading of a middleware adaptor. SAGA middleware adaptors pass jobs to the middleware as its clients and are responsible for low-level communication with it.

The high-level interfaces, provided by SAGA, for communicating with different middleware, remove the need for the use of middleware specific APIs. For example, if an application intends to use three different middlewares then the application has to use a different API set for interacting with each middleware. SAGA, on the other hand, provides a single generic API to interact with every middleware whose adaptor is loaded in the application. Thus, the GP, using SAGA, ‘glues’ a number of client applications together with different middleware using the relevant adaptors. The GP wraps the SAGA API and provides a WSDL [14] binding for client applications.

There are numerous advantages of this approach, compared to more traditional means whereby the client applications directly use the SAGA API implementation. The two most important reasons for adapting a web service based approach to SAGA are as follows:

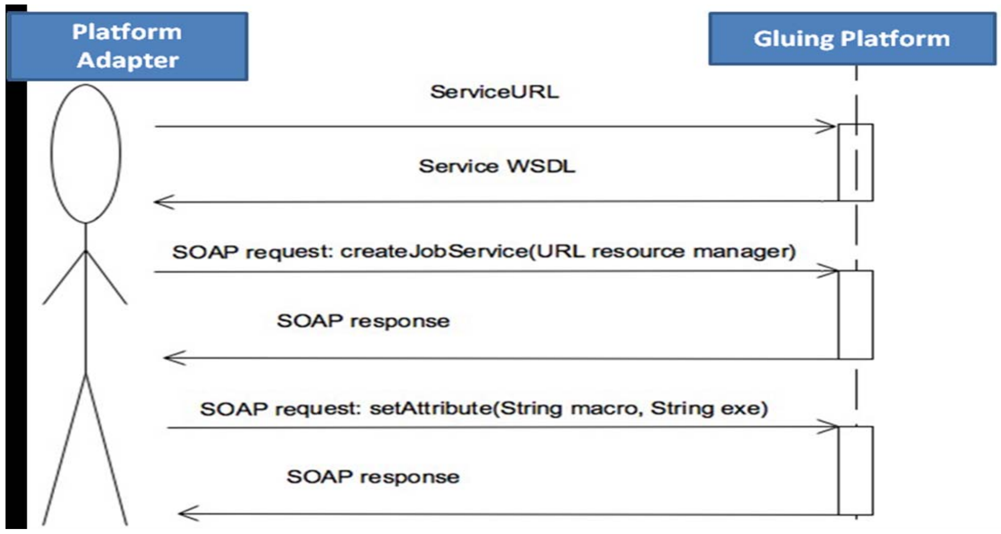
Platform Adapter
Gluing Platform
ServiceURL
Service WSDL
SOAP request: createJobService(URL resource manager)
SOAP response
SOAP request: setAttribute(String macro, String exe)
SOAP response

Figure 2: A sample Use Case

1) SAGA Implementations come packaged in two parts: Middleware adaptors and the API. The client uses the API to communicate to the grid middleware via the appropriate adaptor. In order for the adaptor to communicate with the Grid middleware the client application, SAGA API and the adaptor must be deployed on a system which has the core middleware packages installed, deployed and configured. For instance, consider a user who wants to submit a job to a condor [15] cluster deployed as part of an OMII [16] middleware grid. The client application must be deployed on a node, which has the OMII middleware client installed, and condor installed and configured in submit only mode. If in the future the application is ported to another middleware, the client application must be deployed on a machine which has the new middleware installed. This increases porting costs and introduces complex installation and deployment procedures for applications which make the life of common users quite difficult.

2) SAGA is an evolving standard. Currently there are numerous things which are not provided by the SAGA API. Workflows are one of them and discovery is another. The Gluing Platform can be used as a means to providing functionality, which is not currently provided by SAGA. In this scenario, the Gluing Platform would expose all of SAGA's capabilities. In addition it would also expose higher-level functions which SAGA does not provide, however they deal with the Grid middleware and are required by client services.

The GP shields users from these highlighted difficulties, and deploys the adaptors and middleware at the location of the GP host. Clients interact with the GP via a standard web service based infrastructure. In this scenario, if the middleware needs to be changed, all that the clients require is a new endpoint to an appropriately deployed GP. In order to make the GP SAGA compliant, two separate WSDL descriptions have been provided, one of which points to the full SAGA implementation, and the other describes the extended functionality, which is not currently supported by SAGA. Both WSDL descriptions map to the GP. Additionally, the GP exposes SAGA API functions as web service methods and provides one-to-one correspondence to the SAGA API functions. Client applications can transparently access the GP by using a SAGA SOAP adaptor, which is an implementation of the Adaptor API provided by SAGA. The clients can include the SAGA SOAP adaptor and can write applications using the standard SAGA API classes. The SAGA API calls, generated on the client side, are passed to the GP by the SAGA SOAP adaptor, which is responsible for all communication with the GP.

The SAGA SOAP adaptor is a middleware between the client applications and the GP. The client applications define, create and submit jobs according to the standard specification of SAGA [9]. These instructions are then translated into SOAP requests by the SAGA SOAP adaptor. SOAP requests are used for communication with the Gluing Platform. The Gluing Platform actually executes the client instructions using SAGA APIs and middleware adaptors. The SAGA SOAP adaptor has methods to discover the GP and retrieve its endpoint URL. The endpoint URL is used to access the service WSDL, which is then used for service invocation. The WSDL describes the definition of all the exposed methods. The SAGA SOAP adaptor calls the published methods using SOAP requests and the Gluing Platform sends back SOAP response to the SAGA SOAP adaptor. The SAGA SOAP adaptor then translates the SOAP response and returns the execution results to the client in the form of Java or SAGA specific objects. Figure 2 shows a scenario where a SAGA SOAP adaptor interacts with the Gluing Platform. The SAGA SOAP adaptor passes the middleware and job information to the Gluing Platform using SOAP objects and the SOAP response is sent back to the SAGA SOAP adaptor from the GP.

III. CASE STUDY

The architecture diagram (Figure 3) shows how an application contacts the GP. The pipeline service, shown in the figure, is one of the potential applications of the GP. This service uses the SAGA APIs and SAGA SOAP adaptors to communicate with the GP. The SAGA SOAP adaptor accesses different methods of the GP by getting its WSDL. The methods, published in the WSDL execute the actual instructions which are generated on the client side. Thus, the pipeline service initiates a grid activity which is then forwarded to the GP by the SAGA SOAP adaptor. The GP, as shown in the diagram, can communicate with different Grid middleware such as OMII/GridSAM [10] or gLite etc. This allows client applications, such as pipeline service, to use Grid resources provided through different middleware. The GP uses the JavaGAT [11] middleware, which is a SAGA implementation, and OMII adapters to communicate with the OMII middleware.

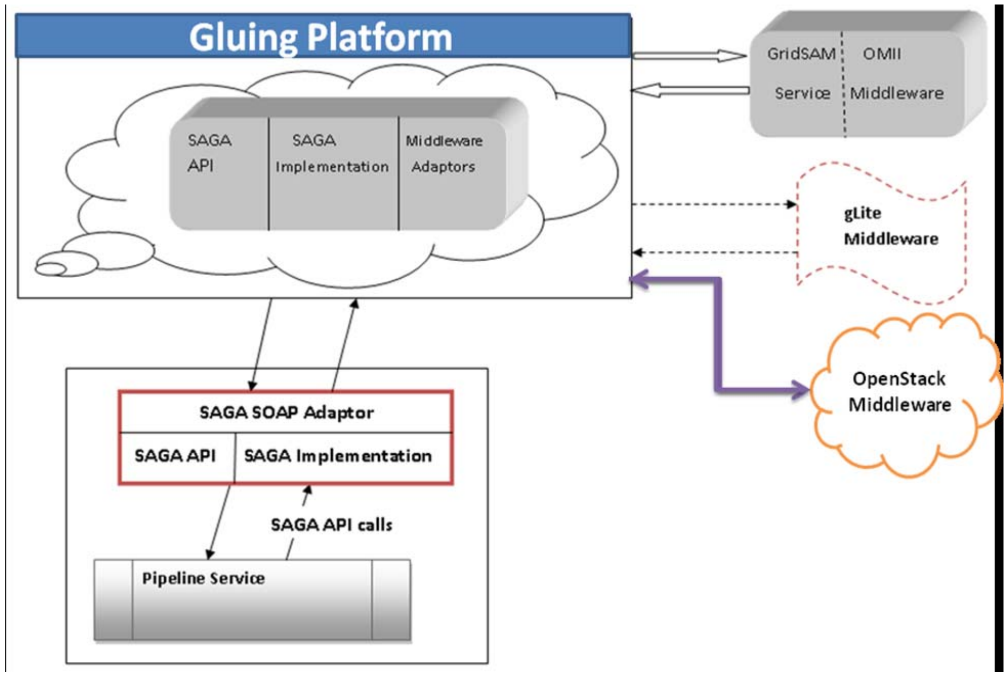

Figure 3: Gluing Platform and Grid Middleware

The GP client applications such as the Pipeline service in the neuGrid project [12], allow clinicians and neuroscientists to create neuro-imaging pipelines, in a user friendly environment, for a series of automated transformations on the brain images. Once the pipelines are constructed, the neuro-imaging pipeline service parallelizes them. The processes defined in the pipelines are usually very compute-intensive and deal with large amounts of data. Therefore, in the next step the workflows are modified for their optimal execution over the Grid. The neuro-imaging pipeline service, using the GP, submits the workflows to the Grid for their execution and the results of execution are returned to the client interface of pipeline service. A simplified design of the pipeline service is shown in figure 4.

The neuro-imaging Pipeline Service passes workflows, or sequences of processes, to the Gluing Platform. The Gluing Platform exploits the SAGA system and executes the processes in the workflows on the Grid using appropriate Grid middleware. The results of execution are passed over to a provenance service and eventually reach back to the client.

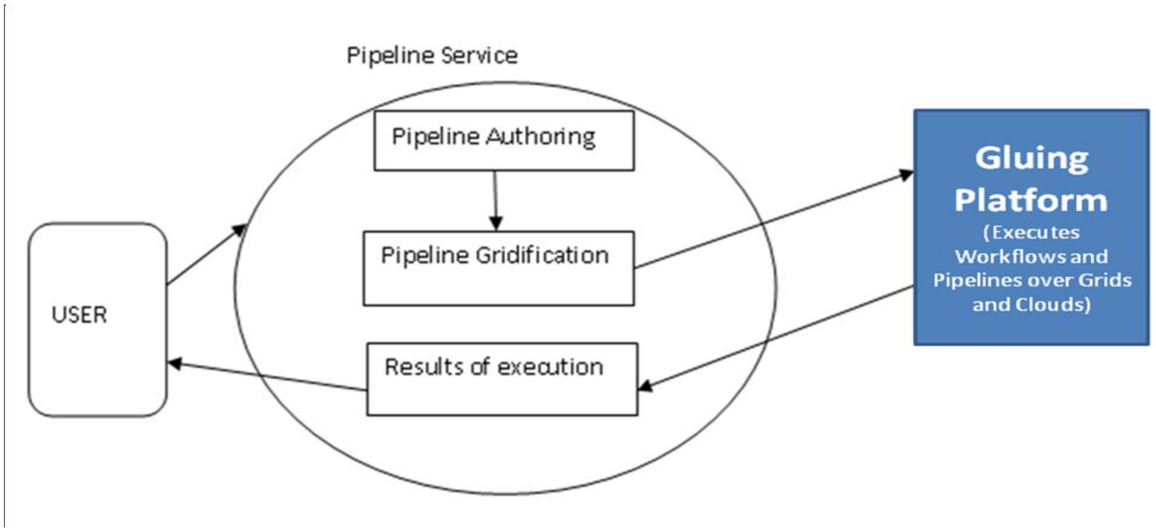

Figure 4: Pipeline Service

IV. Platform Components and Functionality

The GP exposes SAGA APIs in the form of a web service (OR WEB SERVICES) which allows client applications to create, run and monitor jobs transparently from the underlying Grid middleware. The client applications, using the SAGA SOAP adaptor, pass job and middleware information to the Gluing Platform. The platform loads the appropriate middleware adaptor, based on the middleware information. For example if a client wants to run jobs or access data on an OMII based infrastructure, the client specifies an OMII/GridSAM middleware; the GP sets a system property "JobService.adaptor.name" equal to "javaGAT". This system property behaves like an environment variable for the SAGA engine to narrow down the selection of middleware adaptors. The GP then makes sure that JavaGAT loads a GridSAM adaptor by creating an appropriate session and context for SAGA.

The context is a specific piece of information that is shared with a particular session. An application may associate different contexts with a particular session in order to perform different functions in that session.

The GP creates a preferences context for JavaGAT to make sure that it loads the GridSAM adaptor. This is done by setting the context property "ResourceBroker.adaptor.name" equal to "gridsam". This context is then added to the application session for further reference during the life-time of the application. Once the session is created the GP creates a job, based on the job information sent by the client application. The GP uses SAGA job management APIs for creating a job, submitting it to the available Grid resources and retrieving its status. The SAGA job management API covers four classes; these are JobService, JobDescription, Job and JobSelf. The order in which the GP uses this job management and other APIs is as follows:

1) *Selecting a Resource Manager*

The JobService API is used to select a resource manager. An instance of JobService represents a resource manager backend. A resource manager is an endpoint where the job is submitted by the client application. This resource manager can also be an execution service if it executes the job.

Input parameter: To create an instance of JobService class an endpoint URL of resource manager is required as input parameter to *createJobService* method.

Example: The GP uses an endpoint URL for submitting the job to resource manager. For example if resource manager is GridSAM then an instance of JobService is created as: *<JobService> js = JobFactory.createJobService (new URL ("https://HOST:PORT/Gridsam/services/gridsam")*.

2) Job Definition

The *JobDescription* API defines the job using a well-defined set of attributes such as the application executable and associate arguments. The *JobDefinition* attributes behave like tags in JSDL/JDL and thus these attributes mimic JSDL for the middleware and are passed to it internally by SAGA.

Input parameters: The JobDescription API needs two essential parameters in order to define the job i.e. the application executable path and the exact application parameters.

Example: If the application execution is "/bin/echo" which takes a string as input parameter i.e. "hello" then the GP, using JobDescription, defines a job as: *<JobDescription> jd.setAttribute (JobDescription.EXECUTABLE, "/bin/echo")* and *<JobDescription> jd.setVectorAttribute (JobDescription.ARGUMENTS, new String[] {"hello"})*.

3) Data Stagein-out

The job execution results can be stored in an output file as: *<JobDescription> jd.setAttribute (JobDescription.OUTPUT, "outputFile")*. The JobDescription class also defines resources for data staging. For example the results of execution can be staged out at a remote location using FTP as: *<JobDescription> jd.setAttribute (JobDescription.FILETRANSFER, "file://HOST:PORT/outputFile < outputFile")*. The execution results are staged, once execution is complete. The results can be copied by calling the *copy* function of the GP and by passing it source and target URL parameters.

4) Job Creation

The GP creates an instance of the Job class using *createJob* function of JobService, which takes an instance of the JobDescription class as input parameter. Instances of both JobService and JobDescription classes are a pre-requisite for creating a Job.

Example: A job is created as: *<Job> j = <JobService> js.createJob (<JobDescription> jd)*. The GP executes jobs by *<Job>j.run()* and an implementation of *CallBack* interface allows it to get monitoring information of job during the course of execution.

5) Asynchronous Job Execution

The GP also allows the client applications to run different independent jobs at the same time. The feature of running jobs asynchronously is provided by the SAGA *TaskContainer* API. An instance of *TaskContainer* may contain multiple tasks. A task is a SAGA API call, whereas a job is a remotely running application. The SAGA *Job* class extends *Task* interface therefore a *TaskContainer* can also contain multiple jobs. The jobs are executed asynchronously when the instance of *TaskContainer* calls its *run()* method.

Example: If there are two different jobs, j1 and j2 and both copies a file from one URL to another then the *TaskContainer* will add both jobs as: *<TaskContainer> T.add(j1); <TaskContainer) T.add(j2)*. The task container will execute these jobs asynchronously as: *T.run()*.

6) Replica Management

The GP also allows client applications to replicate files over Grid resources using FTP or GridFTP. If client wants to replicate a file to a Grid resource using GridFTP, he can do this by calling *replicate* function of the GP by providing it the location of GridFTP server and the source file.

Example: If the location of the source file is */home/test.txt,* the replica is generated by first creating an instance of for the source file as: *LogicalFile lf = new LogicalFile("/home/test.txt")* and then this file is replicated to a remote server at the server URL as: *lf.replicate(new URL("ftp://server:port/test.txt"))*.

7) Job Monitoring

The GP monitors a job during its lifetime using callbacks, and it provides two ways of retrieving monitoring information. One way is to use the higher level implementation, by which the client application can invoke the *monitor (String jobId)* function of the GP to get complete monitoring information, associated with a particular job id. In this implementation the GP stores the monitoring information of the job in a *String* and it is returned to the client when (s)he invokes the *monitor*. The other way of monitoring a job is to use a SAGA API function. In this implementation the client first gets the job metric and it can then be used to get the metric name and value.

Example: A job metric can be obtained by using a *Job* instance and the *metricName*. If *j* is an instance of *Job* as created in section 7.3.4 and *metricName* is Job.JOB_MEMORYUSE then the job metric can be obtained as: *Metric m = getMetric(j, Job.JOB_MEMORYUSE)*. This job metric contains the information of memory used by the particular job *j*, while it is in execution. The metric value can be obtained by *m.value.*

8) Data Querying

The GP also provides a mechanism for querying the data, staged out or replicated at the remote locations. The staging endpoints may contain a number of execution results in the form of files. A client may want to query a particular pattern to see the list of files present at the remote location. This can be achieved by calling the *find(NSDirectory nsDir, String pattern)* function, exposed by the GP.

Example: The GP queries a particular pattern using the instance of NSDirectory. The instance of NSDirectory points to a remote directory. If the client queries a pattern "*.jpg" to list all the jpg files, staged out at the remote directory then the GP fetches the list as: nsDir.find("*.jpg").

V. CONCLUSIONS AND FUTURE DIRECTIONS

The GP addresses some major requirements of users who wish to adopt a flexible approach to accessing the Grid and Cloud. The heterogeneity of distributed resources and details of Grid or Cloud middleware architectures will be obscured from users. The GP also hides complexities of interfacing with different Grid and Cloud middleware, which will allow access to Grid and Cloud resources through a set of high level functions. This is achieved by exposing SAGA APIs, all communication being achieved through middleware adaptors. Details of middleware interactions are kept hidden from users enabling them to seamlessly use Grid and Cloud functionalities. This shields the low level middleware difficulties from the user and assists them in using resources with little knowledge or interest. The design of the GP is based on service-oriented architecture [17] principles, which will help different services and applications to use service functionalities through standardized interfaces. This will also allow other client applications to use the Platform features by inspecting its WSDL, available online at the service endpoint URL.

Future work includes the provision of support for resource discovery and improving it to a level so that it may be used for production quality needs. Support for workflow composition and enactment is another major issue that needs to be addressed. The support for Cloud platforms such as OpenStack and EUCALYPTUS [13] also needs to be provided, along with support for application hosting, job distribution and concurrency.

VI. REFERENCES

[1] A. Kretsis, P. Kokkinos, and E. Varvarigos. 2009. Developing Scheduling Policies in gLite Middleware. In Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID '09). IEEE Computer Society, Washington, DC, USA.

[2] OpenStack Open Source Cloud Computing Software www.openstack.org/ Last accessed : May 24, 2010.

[3] Keith Jeffery and Burkhard Neidecker-Lutz, The Future of Cloud Computing, opportunities for European Cloud Computing beyond 2010, Expert Group Report public version 1.0, http://cordis.europa.eu/fp7/ict/ssai/docs/executivesummary-forweb_en.pdf.

[4] Kertesz, A., Kacsuk, P., Grid Interoperability Solutions in Grid Resource Management, IEEE Systems Journal, March 2009, Volume: 3 Issue:1 On page(s): 131 – 141 ISSN: 1932-8184.

[5] Rack Space Cloud Hosting, WWW. Rackspace.com, May 24, 2011.

[6] Amazon Elastic Compute Cloud, http://aws.amazon.com/ec2/, May 24, 2011.

[7] Shantenu Jha, Hartmut Kaiser, Andre Merzky, and Ole Weidner, Grid Interoperability at the Application Level Using SAGA. In Proceedings of the Third IEEE International Conference on e-Science and Grid Computing (E-SCIENCE '07). IEEE Computer Society, Washington, DC, USA, 584-591.

[8] Open Grid Forum, http://www.gridforum.org/ Last accessed: May 24,2011

[9] A Simple API for Grid Applications (SAGA), www.gridforum.org/documents/GFD.90.pdf , May 24,2011

[10] William Lee , A. Stephen Mcgough , John Darlington, Performance evaluation of the GridSAM job submission and monitoring system, Proc. 2005 UK e-Science All Hands Meeting, , 2005.

[11] Rob V. van Nieuwpoort, Thilo Kielmann, Henri E. Bal. User-friendly and reliable grid computing based on imperfect middleware. In Proceedings of the ACM/IEEE Conference on Supercomputing, 2007

[12] Alberto Redolfi, Richard McClatchey, Ashiq Anjum, Alex Zijdenbos, David Manset, Frederik Barkhof, Christian Spenger, Yannick Legré, Lars-Olof Wahlund, Chiara Barattieri & Giovanni B Frisoni, Grid infrastructures for computational neuroscience: the neuGRID example, Future Neurology, November 2009, Vol. 4, No. 6, Pages 703-722

[13] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. 2009. The Eucalyptus Open-Source Cloud-Computing System. In Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID '09). IEEE Computer Society, Washington, DC, USA, 124-131. DOI=10.1109/CCGRID.2009.93 http://dx.doi.org/10.1109/CCGRID.2009.93.

[14] Web Services Addressing 1.0 - WSDL Binding, W3C Working Draft 15 February 2005, http://www.w3.org/TR/ws-addr-wsdl, May 24, 2011.

[15] Douglas Thain, Todd Tannenbaum, and Miron Livny, "Distributed Computing in Practice: The Condor Experience" Concurrency and Computation: Practice and Experience, Vol. 17, No. 2-4, pages 323-356, February-April, 2005.

[16] OMII-UK, Software Solution for e-Research, http://www.omii.ac.uk/index.jhtml, Last Accessed: May 24, 2011.

[17] Ashiq Anjum, Peter Bloodsworth, Andrew Branson, Irfan Habib, Richard McClatchey, Research Traceability using Provenance Services for Biomedical Analysis, Studies in Health Technology & Informatics, 2010, Vol 159, pp 88-99 ISBN 978-1-60750-027-8 IOS Press.