Grid Interoperability at the Application Level Using SAGA

Shantenu Jha*†, Hartmut Kaiser *, Andre Merzky‡, Ole Weidner*
* Center for Computation and Technology, Louisiana State University, Baton Rouge, Louisiana, USA, 70803
† Department of Computer Science, Louisiana State University, Baton Rouge, Louisiana, USA, 70803
‡ Vrije University, Amsterdam, Netherlands

***Abstract*— SAGA is a high-level programming abstraction, which significantly facilitates the development and deployment of Grid-aware applications. The primary aim of this paper is to discuss how each of the three main components of the SAGA landscape – interface specification, specific implementation and the different adaptors for middleware distribution – facilitate application-level interoperability. We discuss SAGA in relation to the ongoing GIN Community Group efforts and show the consistency of the SAGA approach with the GIN Group efforts. We demonstrate how interoperability can be enabled by the use of SAGA, by discussing two simple, yet meaningful applications: in the first, SAGA enables applications to utilize interoperability and in the second example SAGA adaptors provide the basis for interoperability.**

I. Introduction

Attempts to make the Grid – a global computational infrastructure a reality [1], have been ongoing for some time. While there exist islands of infrastructure that can be successfully marshaled for a task, the fundamental idea that inspired the "Grid" analogy – indistinguishable computational resources and seamless sharing of resources across administrative, technical and distributed domains is not reality yet.

A lot of effort has been invested in trying to connect these islands[1]. In particular the pioneering efforts of the GIN Community Group[2] at the OGF [3] at identifying and solving the problems related to interoperation are laudable. On the other hand, there have been impressive application-centric efforts to interoperate across distinct Grids, as illustrated in References [4], [5] - to name just a few [2]. These attempts at providing interoperation – that is quick, workable solutions – represent two different approaches to the problem. In the first approach (symbolized by the GIN group efforts), the focus has been on attempting to provide the basic *services* that are required for Grids to interoperate; approaches symbolized by the application groups have been aimed at enabling specific applications to utilize federated Grids. The successful interoperation of Grids in the latter case, is thus specific to the application(s) under consideration, i.e., what might represent interoperation for one application might not represent interoperation for a different application. GIN efforts can be viewed as a bottom-up approach, whilst application-centric efforts as top-down. So far all attempts – whether aimed at providing interoperation at the service-level or application-level – have been quick and short term solutions, in order to begin utilizing Grid resources, for it is not practical, nor desirable that applications wait for federated resources to get every last detail and requirement in place.

e-Science applications must be able to utilize e-Infrastructure – whether sophisticated services, or the next generation networks or customized supercomputers – to perform better, faster and different domain specific research [6], [7]. The question is how can we design and develop e-Science applications, that on the one hand are not limited by the heterogeneity of infrastructure that exists at any given point in time, while on the other hand being immune to the evolving nature of such infrastructure? In other words, how can we compose applications so as to not depend on the underlying infrastructure – either in a dynamical sense (i.e., over a period of time), or different infrastructure at essentially the same time.

We contest that there exists at least one approach that address both: design applications using widely supported, standardized high-level interfaces. It is reasonable to posit that a necessary requirement for applications to work over federated Grids is to abstract out the non-essential details of the specific Grids and one way of effectively doing this is through the use of such standardized, application-level interfaces. Specifically, we suggest that SAGA [8] - which provides simple method calls at the right level of abstraction for the most commonly required Grid-functionality provides most of the application requirements for interoperability.
We follow the working-definitions of interoperation and interoperability as provided by the GIN group [2] and Reference [9]. We extend them to include application level interoperation or interoperability, which to a first approximation, can be thought of as the ability of applications that require explicit Grid functionality to be able to utilize resources across federated Grids. The primary aim of this paper is to discuss and highlight SAGA as a critical enabling technology for such application-level interoperability (ALI). In principle, any sufficiently high-level programming interface with widespread support and commitment to being supported can be used in lieu of SAGA; it just happens to be the case that no such similarly broadly usable interface exist[3]! In section 4, in addition to SAGA enabling interoperability both in principle and in practise, we will discuss how the specific details of our implementation enhance the ability to interoperate across

[1]We consider the following – Grid-of-Grids, Federated Grids and Inter-Grids – to be equivalent and representative of the situation when resources from two or more distinct Grids are utilized. Distinct Grids are those which differ either at the administrative, middleware or service & policy level

[2]The conference series - Challenges of Large Scale Applications in Distributed Environments, provides an interesting sample of application-centric efforts at utilizing federated Grids

[3]If, for example, OGSA would provide an application level interface to OGSA services, which would in fact be supported by the majority of Grid middlewares, there would be no need for an API such as SAGA.

federated Grids.

This paper is structured as follows: In the next section we elaborate further what we refer to as Application-level interoperability (ALI), and distinguish it from service level interoperability/interoperation (SLI). Section III presents the motivation and the basic design principles behind the specification of a high-level programming abstraction such as SAGA; this section also sketches the main packages that are currently specified and implemented. An interesting and pedagogically important part of Section III is a discussion of how SAGA relates to the five current areas of GIN activity. Section IV discusses how good software design principles have been employed in the (first) implementation of the SAGA specification and how it supports application-level interoperability across distinct middleware distributions. Section V discusses a couple of simple, yet interesting applications that we have developed using SAGA, that not only *require*, but also *implement* application-level interoperability. We conclude with a discussion and analysis of our work and some implications for the future of interoperability.

II. Application level interoperability

In this section we will elaborate further on what we mean by application level interoperability, but before that it is important to clarify the type of applications in the scope of this paper. Using a high-level taxonomy, we can classify[4] applications as either Grid-aware or Grid-unaware. Any application can to a first approximation, be thought of as a Grid-aware application if it is cognizant of the underlying distributed infrastructure, and/or for which there is a requirement to explicitly exploit, run or utilize the distributed infrastructure. Simple examples of Grid-aware applications are the class of tightly coupled MPI jobs – which are generalizations of parallel applications to parallelized and distributed applications. For example, MPICH-G2 applications require cognizance of the distributed resources in their RSL description. Additionally, there are "first principles" Grid applications, such as GridSAT [10] and applications which based upon resource aware "learning" algorithms [11], which need to explicitly marshal distributed resources. For these applications the resource utilization is often dynamic and unpredictable; interestingly, the resource requirements and utilization might possibly be dependent upon both the execution trajectory and underlying infrastructure.

Grid-unaware applications, are those that are not cognizant of the environment that they are executed in, and their behavior, resource requirement and possible execution trajectory is *independent* of the resources. Examples include sandboxed applications developed using say BOINC infrastructure, or an application launched in a portal or using a meta-scheduler such as GridWay. Both portals and meta-schedulers qualify as Grid-enabled Programming Environments as shown in Fig.1.

Some defining features of ALI include:

1) Other than compiling on a different or new platform, there are no further changes required of the application

[4]The aim is not to provide rigorous definition, but a classification scheme.

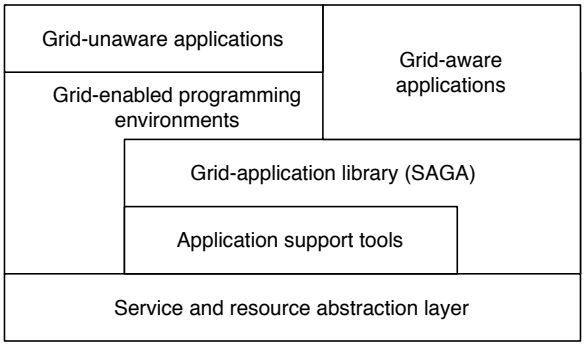

Fig. 1: A high level classification of applications into Grid aware versus Grid unaware. Most Grid-unaware applications can make do with service-level interoperation.This diagram also illustrates the typical mechanisms (and differences) in how Grid-aware and Grid-unaware applications access the service and resource layer.

2) Automated, scalable and extensible solution to use new resources, and not via bilateral or customized arrangements
3) Semantics of any services that an application depends upon are consistent and similar, e.g., consistency of underlying error handling and catching and return

For Grid-unaware applications, it should be easy to see why applications that utilize distributed resources via portals, or by sandboxing or through other Grid-enabled Programming Environments, don't require ALI, but they do require SLI, such as the ability to submit jobs and transfer files directly between middleware distributions. Establishing interoperation/interoperability across Grid-unaware applications is *relatively* easier than to do so for Grid-aware applications. Not surprisingly, interoperability is a necessary condition for any successful Grid-aware application and thus is the real challenge. For the remainder of this paper, we will discuss ALI with a focus on Grid-aware applications. In addressing the relationship between ALI and service-level interoperability/interoperation (SLI), the issue is not that of whether ALI and SLI are orthogonal problems, or if SLI is a necessary condition for ALI, but how does one influence the other. For example, applications based upon the master-client programming model don't really require SLI but do require ALI. It might be obvious that ALI would benefit from SLI, i.e., if SLI is available, ALI is easier to implement, but the converse is not always true, or even intuitive.

The first point almost necessarily implies the need for a high-level, standardized API; as we will discuss such an API provides a necessary condition for ALI, but by itself does not provide a sufficient condition if not implemented or supported completely. ALI can be achieved in various ways: probably the simplest example of ALI comes from the successful deployment of MPICH-G2 based applications, which in turn requires simply extending the MPI library to an MPICH-G2 library where appropriate. Also as alluded to, high-level interfaces that provide functionality beyond messaging are useful. In addition to the above three requirement, there exist features that would be *nice to have*, as

they would make interoperability more robust and extensible. Such features include performance estimates/guarantees, or service-level agreements (SLA) between resource providing Grids. For example, there are Grid-aware applications for which, interoperability is predicated on the ability to federate resources from distincts Grids synchronously, which in turn requires features such co-scheduling, or at the very least a Quality of Service (QoS) assurance or SLA that facilitate this. In such cases ALI requires consistency and agreements at the policy level (going beyond firewalls); this is unlikely to be the case for SLI. Not having these agreements will lead to adhoc arrangements, say over the phone and system administrator exchanges, and which at best, can lead to interoperation. Such interoperation has been shown to work for tightly-coupled applications such as NeKTAR and Vortonics[5], but it is also realized that such solutions are not scalable.

Finally and to state the obvious, for both Grid-aware and/or Grid-unaware applications, there needs to be operating-system level and middleware level support for whatever framework the application is developed on. Revisiting the MPICH-G2 application for example, the platform should not only provide a local implementation of MPI, but also compatible Globus services; however, much of this requirement is not specific to distributed computing hence we do not elaborate further.

III. SAGA

The Simple API for Grid Applications (SAGA) is an API standardization effort within the Open Grid Forum (OGF) [12] – an international committee that coordinates the standardization of Grid middleware and architectures. SAGA provides a simple, POSIX-style API to the most common Grid functions at a sufficiently high-level of abstraction so as to be able to be independent of the diverse and dynamic Grid environments. SAGA has been often been referred to as the MPI [13] for Grid Programming, in that is a simple, high-level programming abstraction that provides most of the functionality. In addition to being simple, a noteworthy and critical feature of SAGA is that it is on the road to becoming a community standard, thus strengthening the analogy with MPI. The interface defined by the SAGA specification is grouped as a set of functional packages, which we discuss in this section. Version 1.0 [14] of the specification has been submitted to the OGF editorial pipeline and is currently under review.

A. SAGA Packages:

The SAGA packages are as follows:

- File package - provides methods for accessing local and remote filesystems, browsing directories, moving, copying, and deleting files, setting access permissions, as well as zero-copy reading and writing
- Replica package - provides methods for replica management such as browsing logical filesystems, moving, copying, deleting logical entries, adding and removing physical files from a logical file entry, and search logical files based on attribute sets.
- Job package - provides methods for describing, submitting, monitoring, and controlling local and remote jobs. Many parts of this package were derived from the largely adopted DRMAA [15] specification.
- Stream package - provides methods for authenticated local and remote socket connections with hooks to support authorization and encryption schemes.
- RPC package - is an implementation of the GGF GridRPC API [16] definition and provides methods for unified remote procedure calls.

The two critical aspects of SAGA are its simplicity of use and the fact that it is well on the road to becoming a community standard. Simplicity arises from being able to limit the scope to only the most common and important grid-functionality required by applications. There a major advantages arising from its simplicity and imminent standardization. Standardization represents the fact that the interface is derived from a wide-range of applications using a collaborative approach and the output of which is endorsed by the broader community.

B. SAGA In Relation to GIN:

Having outlined SAGA, its features and design goals, we now discuss its relationship to the (five) areas of GIN.

1) SAGA and Information Services and Modeling: The GIN effort for interoperation of various information services focuses mainly on the identification of common information elements, and of schema translators and data providers (adaptors) to access these information elements. That additional layer on top of the individual middleware solution provides a GLUE based representation of the individual information resources, via a BDII-based (i.e. LDAP based) infrastructure.

SAGA on the other hand provides the application level access mechanisms (API, storage management, query language) which allows applications to easily access and use Grid resource and application information. SAGA exposes, however, only those information elements which are of interest to the applications (i.e. does not expose information which are *internally* used for scheduling decisions). Additionally, SAGA allows the transparent management of custom information items within the same framework, which enables Grid-aware applications to use the information service paradigm to persistently store, search and retrieve application level information. The resource level information used by GIN for SLI are thus extended by application level information used for ALI.

2) SAGA and Job Submission and Management: GIN, and indeed many recent and relevant Grid projects, rely on OGSA-BES (OGSA Basic Execution Service) and JSDL (Job Submission and Description Language) for interoperation of job submission and job management. BES and JSDL can indeed be used on a wide variety on Grid middleware distribution, as both standards are intentionally easy to map to other, existing job description languages and submission systems.

The SAGA job management API exposes the same basic functionality as BES, and provides additional program level

simplifications and abstractions to the application. For example, the id of a submitted job in SAGA, is represented as an object, so that actions on that job can be performed directly, without the need to explicitly contact the respective job service.

Incidentally, the job state model exposed by SAGA is exactly the same as the job state model defined by BES. This is not a coincidence, as the state model was designed by the SAGA and BES groups at the OGF together. Due to the two-level state-model, it can, (a) accommodate all higher level states which SAGA calls and BES actions can actually act upon, and (b) can represent all additional states any specific job management system may internally use.

The SAGA Job description is represented by a subset of JSDL and JSDL-SPMD keywords, and has some additional keywords which are used to allow for application level job scheduling. Even though JSDL is not supported explicitly, the application level job descriptions are thus easily mapped to JSDL documents, and the SAGA implementation can transparently use the BES/JSDL layers to submit and manage the jobs. For application level interoperation it is important to mention that jobs created from within an application can be controlled, communicated with, and reaped by that application: this allows for extremely simple application level implementations of job cloning, spawning, and task farming scenarios. Also, SAGA allows easy bootstrapping of distributed applications, as job IDs can easily be exchanged over the information service interfaces mentioned before.

3) SAGA Data Management and Movement: The GIN community identified [9] GridFTP [17] as *"lowest common denominator for data movement in Grids today"*. Other, more advanced data movement infrastructures, such as various Grid storage managers or replica systems, are assumed to form *islands*, i.e. disconnected sets of resources which do not (easily) allow the interchange of data sets.

The SAGA approach differs in that it completely hides the details and differences of all data movement and data management infrastructures. As an API, SAGA cannot really provide the interoperability the infrastructure seem (according to GIN) to be inherently missing, but can at least free the application programmer from the need to differentiate at a programmatic level. For example, SAGA allows to specify the location of a file as '`any://remote.host.net/dir/file.typ`'. The pseudo-schema `any` allows the SAGA implementation to automatically choose a suitable schema, depending on the deployed middleware. That mechanism cannot, of course, guarantee that the file is indeed available via (literally) any schema or protocol, or that the available schema provides the full scope of access mechanisms defined by SAGA, but the application programmer is nevertheless freed from the need to explicitly switch the application code for each island.

4) SAGA Authorization and Identity Management: "Identity management and authorization of resource usage should, be performed transparently at the application level" is a repeatedly stated end user requirement, and the SAGA API follows that guideline. It exposes only minimal interfaces for identity management, and most authentication and authorization processes are not exposed at the application level. That allows to interface a SAGA implementation to a wide variety of security approaches, including of course GIN-AUTH/GIN-VOMS services.

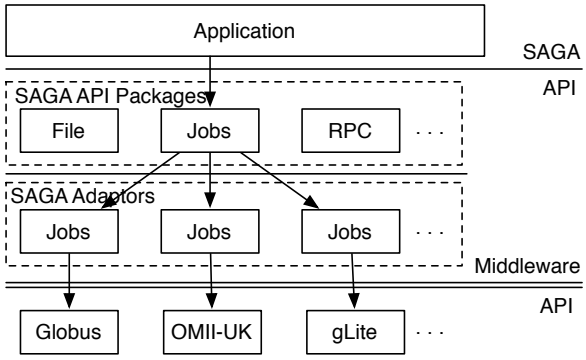

Fig. 2: An example of how SAGA is an effective mechanism for Grid Interoperability for job submission.

5) SAGA and Cross-Grid Applications: SAGA as an API cannot, by design, provide service level Grid interoperability. SAGA implementations can, however, very well complement the GIN efforts at the application level, as they allow applications to be *easily* written and to *seamlessly* utilize the infrastructure. With SAGA, the same application code can, (a) run on multiple Grids, and (b) interoperate with applications running on other Grids, thus allowing for truly interoperable Grid aware applications..

C. SAGA and ALI:

The previous section discussed how SAGA relates to the five areas of GIN. In this brief section, using the example of job launching, we will outline how SAGA provides ALI. Figure 2 shows how the appropriate adaptors (i.e. middleware bindings) are invoked by the SAGA engine in order to provide correct execution on different Grid middleware distributions. The basic architecture and the mechanism are the same for other (functional) packages (e.g. Files and RPC). We will provide details in the next section about the design and intelligence required for SAGA to implement this properly. But as shown in Fig. 2 the relevant point here is, SAGA provides ALI by having the same function calls from within the application but, with the appropriate execution and different implementation within adaptors on varied middleware distributions. Thus in addition to the implementation of the core engine, SAGA's ability to provide ALI is as good as the adaptor set for the middleware distribution that it aims to work on. As each adaptor usually is aimed at connecting to a specific middleware and all Grid related API calls are dispatched by the adaptors to the appropriate middleware functionality, the adaptor quality and their syntactic and semantic uniformity with regard to the SAGA specification is of central importance for ALI.

SAGA as it stands can provide most, but not all requirements for ALI as outlined in Section II. Specifically, it cannot provide features such as QoS, co-scheduling and currently

does not have support for SLA. We contest that this is not an intrinsic limitation in the design of SAGA, but a design decision to ensure the "S" (for simple) in SAGA as well as reflection of the complexity of solving implementation details of co-scheduling across federated Grids.

IV. SAGA Implementation and how it supports Interoperability

There are at least two ongoing independent implementations of the SAGA specification (this is a necessary requirement for an OGF specification to become a standard) – one in Java and the other in C++. In the remainder of this paper, we will refer to the SAGA C++ reference implementation as the "implementation". The C++ implementation is being developed in close conjunction with the OGF standard and aims for a complete and strict adoption of the described interfaces. Due to characteristics of Grids, an implementation must cope with a number of dynamic requirements. Some of the major requirements of any implementation in order to ensure flexibility and interoperability are:

- The implementation must be portable and, both syntactically and semantically, platform independent.
- It must be amenable to evolving Grid standards and changing Grid environments.
- It must be able to cope with future SAGA extensions, without breaking backward compatibility.
- It must shield application programmers from the evolving middleware, and it should allow different Grid middleware distributions and versions to co-exist.
- It must actively support fail safety mechanisms, and hide the dynamic nature of resource availability.
- It must meet other end user requirements outside of the actual API scope, such as ease of deployment, ease of configuration, documentation, and support of multiple language bindings.

It is interesting to note that the design objectives are all consistent with ALI; almost as if SAGA were designed to provide ALI by design.

A. The Overall Architecture

Although the Simple API for Grid Applications is, by definition *simple* for application developers, this doesn't imply that the implementation itself will be simple. A major effort was made to build as much logic and functionality as possible into the SAGA library core, providing all the needed common functionality for the functional packages. To understand the features of the SAGA implementation, it is useful to present the library components along three orthogonal dimensions – horizontal, vertical and feature-level extensibility. For purposes of interoperability, we believe the vertical extensibility feature is the most significant.

1) Horizontal Extensibility – API Packages: The SAGA specification is object oriented and defines a set of API groups keeping objects of related functionality together (packages). Our implementation uses this functional grouping to define *API packages*. Current packages are: file management, job management, remote procedure calls, replica management, and data streaming. Each of these packages constitutes a separate and independent module. These modules depend only on the SAGA engine, the user is free to use and link only those modules actually needed by the application, minimizing the memory footprint. Each of these packages benefits from features of the core engine that promote interoperability such as middleware independence.

2) Extensibility for Optimization and Features: Many features of the engine module are implemented by intercepting, analyzing, managing, and rerouting function calls between the API packages, (where they are issued) and the adaptors (where they are executed and forwarded to the middleware). To generalize this management layer, a PIMPL [18] (Private Implementation) idiom was chosen, and is rigorously used throughout the SAGA implementation. This PIMPL layering allows for a number of additional properties to be transparently implemented, and experimented with, without any change in the API packages or adaptor layers. These features include:

- generic call routing
- task monitoring and optimization
- security management
- late binding
- fallback on adaptor invocation errors
- latency hiding mechanisms

The decoupling of these features from the API and the adaptors succeeds because, these properties affect only the IMPL side of the PIMPL layers. The engine module is completely generic, and loosely coupled to both the API and adaptor layers. Any changes to the engine, optimizations, latency hiding techniques, monitoring features etc., can be implemented in the engine, and do not influence the API and adaptor extensions. This facilitates interoperability because it minimizes the dependency on a concrete middleware bound to a given SAGA API object.

3) Vertical Extensibility – Middleware Bindings: A layered architecture (see figure 2) allows us to vertically decouple the SAGA API from the used middleware. Separate adaptors, either loaded at runtime, or pre-bound at link time, dispatch the various API function calls to the appropriate middleware. Usually there will be a separate set of adaptors for each type of supported middleware. These adaptors implement a well defined *Capability Provider Interface* (CPI) and expose that to the top layer of the library, which makes it possible to switch adaptors at runtime and hence switch between different (and even concurrent) middleware services providing the requested functionality. The top library layer dispatches the API function calls to the corresponding CPI function. It additionally contains the *SAGA engine* module, which implements:

- core SAGA objects such as session, context, task or task_container – these objects are responsible for the SAGA look & feel, and are needed by all API packages;
- common functions to load and select matching adaptors, to perform generic call routing from API functions to the selected adaptor, to provide necessary fall back implementations for the synchronous and asynchronous

variants of the API functions (if these are not supported by the selected adaptor).

The dynamic nature of this layered architecture enables easy future extensions by adding new adaptors, coping with emerging grid standards and new grid middleware, enabling interoperability.

B. Generic Call Routing

The core SAGA engine has the built in ability to generically route SAGA API method calls to middleware adaptors. This is one of the central features that enable our implementation with interoperability, because it provides a generic means of selecting proper middleware services based upon the specific application environment, execution context, and user preferences. The essential idea of this routing mechanism is to represent any SAGA API call as abstract objects, and to redirect their execution depending on several attributes and the availability of suitable adaptors. For example, an asynchronous method call for a `saga::file` instance is preferably directed to a asynchronous file adaptor, or, if such is not available, to a synchronous file adaptor (the method gets executed in a thread then, making it asynchronous to some extent), or, if that is not available either, returns an error (`NotImplemented`).

This routing mechanism allows for:

- trivial (synchronous) adaptor implementations,
- late binding: a different adaptor can be selected for each call, even on the same API object instance,
- variable adaptor selection strategies, e.g. based on adaptor meta data, user preferences and heuristics,
- latency hiding, e.g. by clustering related method calls (bulk optimization), or by automatic load distribution over multiple adaptors (not implemented yet).

Figure 3 depicts the point in the sequence of calls where this call routing mechanism is injected by the SAGA engine.

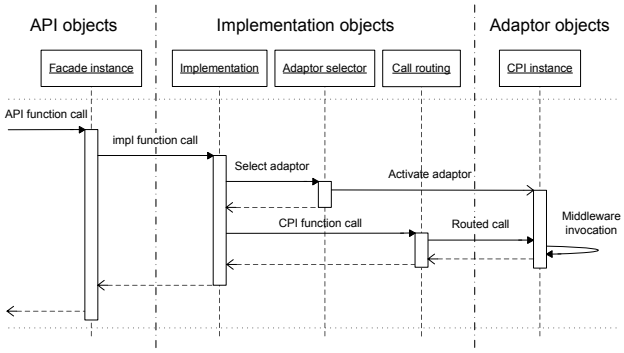

Fig. 3: API function call: Diagram illustrating the execution sequence through the different object instances during a call to any adaptor supplied function. The generic implementation of this call routing in the SAGA engine enables interoperability because it provides the means of uniform selection of proper middleware services based upon the specific application environment, execution context, and user preferences.

As stated above, in terms of interoperability the selection of suitable adaptors at runtime is a central feature in our library implementation (see figure 3). Conceptually this is a simple mechanism: on loading, the adaptor components register their *capabilities* in the adaptor registry. If a method is to be executed, the adaptor selector searches that registry for all adaptors implementing that methods capability. All suitable adaptors are then ordered (best/most suitable first), and are tried one-by-one, until the method invocation succeeds. The adaptor selection again is routed through SAGA engine components, generically implementing this for any function to be routed to a CPI instance.

C. Lessons Learnt

This section will summarize the most important implementation properties form the interoperability standpoint:

- *Uniformity over Programming Languages:* Our implementation follows the SAGA API specification closely. It is also designed to accommodate wrappers in other languages, to provide the same semantics, and similar look & feel to other language bindings. A Python wrapper for our library is in alpha status, and we plan to add similar thin wrappers to provide bindings to C, FORTRAN, Perl, and possibly others.
- *Adaptability to Heterogeneous and Dynamic Environments:* Flexible adaptor selection, late binding, and fall back mechanisms allow for additional resilience against a dynamic and evolving run time environment.
- *Modularity makes the Implementation Extensible:* The adaptor mechanism allows for easy extensions of the library, to provide additional middleware bindings.
- *Portability and Scalability:* Our library implementation is in fact very portable, as we strictly adhere to the C++ standard and portable libraries. In fact, we currently develop the library on Windows, Mac and Linux concurrently, so we are confident that we are able to cover the three major target platforms without any problems.

V. Demonstrating Interoperability

We will outline two applications that demonstrate the connection between SAGA and ALI. In the first application, we show how a SAGA based application exploits ALI in order to implement its functionality; in the second example, we show how SAGA provides simple implementation of file access and management for different applications.

A. Network Performance Aware Application:

We being with a discussion of a network-centric application, that is capable of acquiring application-specific network characteristic data, determining *ideal* migration target based on network characteristics and then to migrate itself across heterogeneous Grids without any changes at the application level. We demonstrate how it can be used for gathering network characteristics for a Poisson Equation solver; details of the application can be found in Ref. [19].

The general architecture of the model application is based on the programming abstraction provided by SAGA and the Cactus framework citeX0. A set of thorns that provide specific functionality are orchestrated by the Cactus flesh as depicted

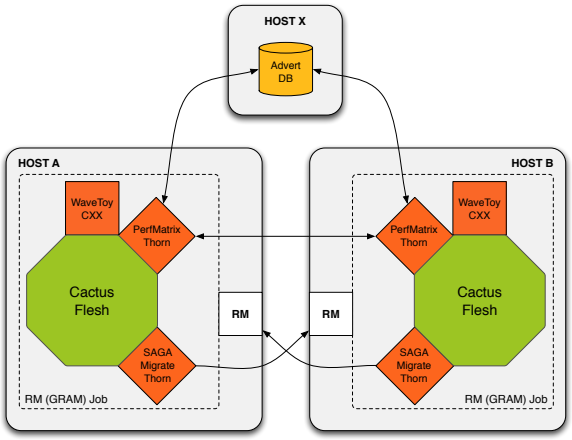

Fig. 4: The components of the application. The Cactus framework orchestrates the *WaveToy* thorn as an example for a distributed calculation, the *NetPerf* thorn for network performance measurement, and the *SAGA Migration* thorn for replication and migration. The *Advert DB* stores and archives the measurement results. Note that the application runs under the control of a *Resource Manager (RM)* using SAGA's job management package.

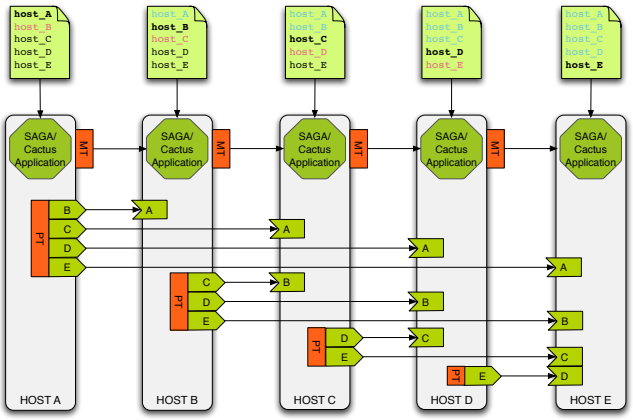

Fig. 5: The algorithm used by the PerfMatrix thorn. Based on a list of resources, the application spawns itself, launches Netperf endpoints and measures the throughput of all possible connections.

in Fig. 4. With the SAGA functionality available it was easy to implement Cactus thorns that can interact with remote resource mangers (e.g. Globus GRAM2), copy, read, and write files from and to remote locations (e.g. Globus GridFTP), and access a remote PostgreSQL based advert service as logging and storage facility. We refer the reader to Ref. [19] for a detailed discussion of the architecture, implementation and description of the individual thorns; here we will discuss mainly the PerfMatrix thorn, which takes care of the intrinsic network performance measurement and persistent storage of the results. *PerfMatrix Thorn:* The PerfMatrix algorithm uses a list of computational resources which are potential migration targets for the application. After starting up, the initial application spawns itself onto all available hosts. Once all jobs have been launched, the original spawning application first establishes netperf [20] connections with all the spawned applications; this is followed by the spawned applications establishing netperf connections amongst each other, following the scheme shown in Fig. 5. Job spawning, control, and I/O redirection is done entirely using SAGA's job management package. Once a netperf process returns a throughput result, the PerfMatrix thorn uses the SAGA advert-service package to announce the result to a central database. After all netperf processes have finished and published their results, the database contains a host-to-host throughput performance matrix along with a timestamp which is available to other thorns as well as other applications.

SAGAMigrate The SAGAMigrate thorn is a Cactus thorn written in C++ that uses SAGA functions (for example `file.copy` and `job_description.create_job`) to perform a simulation migration. SAGAMigrate copies a restart parameter file and the checkpoint file(s) from one machine to another (for example using the SAGA Globus adaptors).

An interesting feature of this application is *the ability to separate the computational logic from the distributed logic*. For example, jobs on different resources, establish connections pairwise and collect information which is published to an advert service through SAGA's job-management package, whilst the PerfMatrix thorn remains independent of the distributed aspects of the set of netperf end-points. This arises from using the correct abstractions (SAGA and Cactus) and enables the application to be easily generalized to more complex network performance requirement scenarios. This application can be deployed on any resource independent of the middleware stack. All that is required is support for SAGA *adaptors* and the corresponding client side middleware bindings. Thus beyond compiling the application code on the other resources, there is no need for the application to know about the details of the middleware or platform details of the remote resources, whilst all along working across distinct Grids!

B. Seamless Integration of Heterogeneous Grid Resources using SAGA and FUSE

Another demonstration of how SAGA can enable seamless data management and movement across different Grids, is the implementation of a filesystem driver. The filesystem driver is written using the FUSE (Filesystem in UserSpace) [21] library and allows any remote filesystem supported by a SAGA adaptor to be mounted onto a local directory. This driver uses SAGA to access remote filesystems through different services, such as GridFTP or SSH. Figure 6 shows the architecture of this application. Other filesystem types can be seamlessly added just by writing appropriate SAGA adaptors. Since the filesystem drivers are integrated into the operating system, it makes the remote files available to any application running on this system.

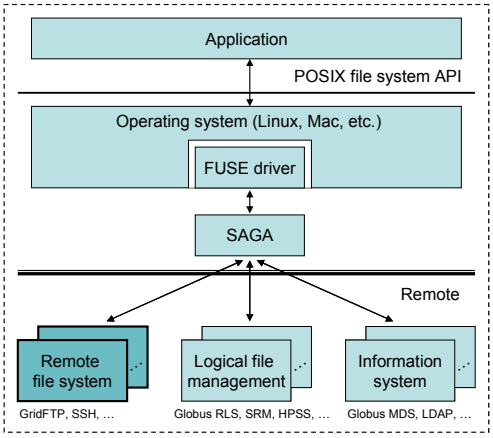

Fig. 6: Any application can access remote files by using the POSIX filesystem API. The operating system dispatches these calls to the FUSE driver, which in turn uses SAGA to execute the required operation.

VI. Conclusion

As efforts by the GIN community-group and other application-specific groups have shown, interoperation is a hard problem to solve and interoperability even harder! In this paper we discuss how each level of the SAGA landscape – interface specification, the engine and middleware specific adaptors - contributes to providing application level interoperability. There are many applications that need to use federated Grids [5], [9], and utilizing SAGA to develop the Grid functionality of these applications provides an effective way to do so.

On the other hand, if the development and deployment of applications across federated Grids is to be facilitated, support and development for SAGA adaptors for different middleware needs to be forth-coming and self-sustaining and will thus require explicit support, from both the middleware developers and resource providers. Although, it might be debatable as to when exactly a critical mass of the community threw their support behind MPI, but irrespective of the timing, it is clear that the technical merits of MPI coupled with the fact that it was a community standard played a great role in increasing the number of applications developed using MPI and the number of vendors that were willing to support it – thus providing MPI-based applications the ability to interoperate across different parallel platforms. As SAGA becomes a standard [22], [23], [14] there will be increased willingness on the part of middleware developers and resource providers to provide these adaptors to the application community. Additionally, as the deployment of SAGA and required adaptors becomes wider (pervasive) and deeper (greater functionality), there is an increased incentive for application developers to use SAGA.

Making the global infrastructure a reality, critically depends upon the availability of computational infrastructure that provides resources that can be seamlessly used; developing both applications and tools using SAGA is an effective mechanism for ensuring such interoperability.

VII. Acknowledgements

This work has been made possible thanks to the internal resources of the Center for Computation & Technology (CCT) at Louisiana State University. Important funding for SAGA specification and development has been provided by the UK EPSRC grant number GR/D0766171/1 (via OMII). We would like to acknowledge the support of the OGF SAGA Research Group for their work towards developing the SAGA specification. We thank Yaakoub El Khamra and others from the Cactus team; we also thank Chris Miceli - an undergraduate student in the SAGA group for working on SAGA validation tests and the SAGA-FUSE project. Finally, SJ acknowledges the e-Science Institute, Edinburgh for supporting the theme, "Distributed Programming Abstractions".

References

[1] Fran Berman, Geoffrey Fox and Tony Hey, "Grid Computing: Making the Global Infrastructure a Reality," doi 10.1002/0470867167, http://dx.doi.org/10.1002/0470867167 Wiley Series 2003.
[2] GIN Community Group http://forge.ogf.org/sf/projects/gin.
[3] Open Grid Forum http://www.ogf.org.
[4] G Allen, et al, "Supporting Efficient Execution in Heterogeneous Distributed Computing Environments with Cactus and Globus", in Proceedings of Supercomputing 2001, Denver, USA, 2001.
[5] Bruce Boghosian et al, "Nektar, SPICE and Vortonics – Using Federated Grids for Large Scale Scientific Applications, Proceedings of Challenges of Large Applications in Distributed Environments (CLADE) 2006," ISBN 1-4244-0420-7.
[6] Tony Hey et al, UK e-Science Programme: Next Generation Grid Applications, 2004, International Journal of High Performance Computing Applications, Vol. 18, No. 3, 285-291 DOI: 10.1177/1094342004046054.
[7] Tony Hey et al, "Cyberinfrastructure for e-Science", Science, Vol. 308, no. 5723, 2005. http://www.sciencemag.org/cgi/content/abstract/308/5723/817.
[8] SAGA - A Simple API for Grid Applications, http://saga.cct.lsu.edu/.
[9] Interoperation Scenarios of Production e-Science Infrastructures, Morris Reidel et al, submitted to Third IEEE International Conference on e-Science and Grid Computing (2007).
[10] W. Chrabakh and R. Wolski, "GridSAT: A Chaff-based Distributed SAT Solver for the Grid," *SuperComputing*, vol. 00, p. 37, 2003.
[11] F. J. Seinstra and J. M. Geusebroek, Color-Based Object Recognition on a Grid, Proceedings of the 9th European Conference on Computer Vision (ECCV 2006) , Graz, Austria, May 7-13, 2006, Springer-Verlag.
[12] Open Grid Forum. [Online]. Available: http://www.ogf.org/
[13] MPIforum: http://www.mpi-forum.org/.
[14] Tom Goodale et al, "A Simple API for Grid Applications," To be published as Grid Forum Document GFD.90, 2007, Open Grid Forum.
[15] Distributed Resource Management Application API, http://drmaa.org.
[16] http://forge.ogf.org/sf/projects/gridrpc-wg.
[17] W. Allcock, J. Bester, J. Bresnahan, S. Meder, P. Plaszczak, and S. Tuecke, "GridFTP: Protocol extensions to FTP for the Grid," *GWD-R (Recommendation)*, Apr. 2003.
[18] H. Sutter, "Pimples–Beauty Marks You Can Depend On," *C++ Report*, vol. 10, no. 5, 1998, http://www.gotw.ca/publications/mill04.htm.
[19] S. Jha et al, "Design and Implementation of Network Performance Aware Applications Using SAGA and Cactus", submitted to Third IEEE International Conference on e-Science and Grid Computing (2007).
[20] NetPerf Homepage, http://www.netperf.org/netperf/NetperfPage.html.
[21] FUSE: Filesystem In Userspace, http://fuse.sourceforge.net/.
[22] Shantenu Jha et al, "A Collection of Use Cases for a Simple API for Grid Applications," Grid Forum Document GFD.70, 2006. [Online]. Available: http://www.ggf.org/documents/GFD.70.pdf
[23] S. Jha et al, "A Requirements Analysis for a Simple API for Grid Applications," Grid Forum Document GFD.71, 2006. [Online]. Available: http://www.ggf.org/documents/GFD.71.pdf