Encyclopedia of Grid Computing Technologies and Applications
SAGA – The Simple API for Grid Applications – Motivation, Design and Implementation

Authors:
Shantenu Jha (Louisiana State University, Baton Rouge, Louisiana, USA), sjha@cct.lsu.edu
Hartmut Kaiser (Louisiana State University, Baton Rouge, Louisiana, USA), hkaiser@cct.lsu.edu
Andre Merzky (Louisiana State University, Baton Rouge, Louisiana, USA), andre@merzky.net
John Shalf (Lawrence Berkeley National Lab, USA), JShalf@lbl.gov

Motivation

A key impediment to accelerated development of Grid applications and consequently the uptake of Grids is the scarcity of high-level application programming abstractions that bridge the gap between existing grid middleware and application-level needs. Application developers are daunted by the complexity of the vast array of low-level Grid and distributed computing software APIs that currently exist; APIs have traditionally been developed using a bottom-up approach that exposes the broadest possible functionality. Coding using these bottom-up APIs requires extremely verbose code to implement even the simplest of application-level capabilities. Many Grid computing projects [**?**, **?**, **?**] recognized the need for higher-level programming abstractions to simplify the use of distributed computing for application developers. The Simple API for Grid Applications (SAGA) attempts to consolidate community effort and make ends meet by employing a top-town approach to distributed computing software infrastructure.

The specification and implementations of the SAGA API has been guided by detailed examination of the requirements expressed by existing and emerging distributed computing applications in order to find common themes and motifs that can be reused across more than one use-case. The main governing design principle for the SAGA API is the 80:20 Rule: "Design an API with 20% complexity that serves 80% of the application use cases". They are intended to cover the most common application-level distributed computing programming constructs such as file transfer, and job management. In general, SAGA embodies the most commonly required features derived from a broad survey of the community and provides the most common grid programming abstractions that were identified by several use cases.

Distributed environments are required for a range of reasons (higher throughput, lower times-to-solution, qualitatively different resources, more-of- the-same resources etc.) and in different usage modes. Some applications need to be designed – programmatic or otherwise – in order to be able to utilise the distributed heterogeneous resources. We refer to such applications as "distributed by design". On the other hand there are applications that are not cognizant of the environment they are deployed in – distributed or localized. Consequently, there are no changes required of the application independent on the environment they operate in; such applications can be considered "distributed by deployment". SAGA provides the abstractions for both classes of applications. A representative example of the latter class, are applications that need to be task-farmed across a range of resources (local or distributed); SAGA provides a programming mechanism that enables the creation of a task-farm manager which is then responsible for marshalling all resources, file-staging etc., and thus enables applications to remain oblivious to the details of the environment (middleware distribution, resource access and control mechanism etc.). For applications that are distributed by design, SAGA provides the ability to exploit the flexibility and power that comes with operating in a distributed environment, without the need for the application to be specifically adapted for every different resource encountered; this is done by providing a consistent, uniform and simple programmatic interface to orchestrate a wide range distributed resources. For example, applications that need to query a range of resources at run time, to determine the optimal resource to spawn a sub-task onto, benefit from the ability to do so programmatic using standardised interfaces such as those provided by SAGA.

By definition Grids are characterised as dynamic and heterogeneous environments. They are dynamic due to time-dependent resources loads, availability and access patterns; the aggregation of specialised resources in

different administrative domains is one source of the heterogeneity. Applications that are designed for dynamic and heterogeneous environments require the ability to manage varying levels of heterogeneity and dynamical resources. SAGA is the first comprehensive attempt to provide a programmatic approach for the development of applications so as to utilise distributed environments, either by design or by virtue of deployment. In addition to simplifying the programming environment for application developers, SAGA insulates applications from technological, version changes and other low-level implementation details that regularly occur in the lower layers of the software stack.

SAGA: Addressing Distributed Application Requirements

Although there are a bewildering number of specific functionality that distributed applications require, a majority of them require a small subset of these functionality. Consequently the scope of SAGA could potentially be very broad; thus a design decision was required about how to balance functionality and simplicity.

SAGA covers the most commonly occurring (general) requirements; this defines the target space of the API. The general requirements in addition to driving specific design aspects of the interface specification (modular, language and middleware independent etc.) also influence the implementation of specification.

General Requirements: The main objective of the SAGA API effort is to **enable programmers of scientific applications to utilize Grid environments**, without the need to understand all details of that environment. With that goal, SAGA has to:

- be very simple, i.e. the API should hide all non-essential features, unless they are explicitly needed;
- shield application development from the complex, diverse and evolving Grid middleware;
- be extensible and future proof, i.e. the API should be able to accommodate additional use cases and future Grid programming models;
- provide the abstraction from which Grid programming models can be constructed;

Functional Requirements: The functional requirements of the SAGA API and its scope are drawn from a set of use cases[**?**]. These use cases were comprised of a range of applications such as requiring multi-physics multi-component distributed simulations, high-end visualization, computational steering. A high level summary of the requirements of the applications that motivated the SAGA effort is shown in Table 1 (more details can be found in the use SAGA Requirement document[**?**]). The list represents the functional scope of the SAGA API, as it defines what functionality the API should cover: for each package listed there, a number of programming models and API methods included in the SAGA API.

Functional Area	#
Job Management	16
Resource Management	13
Data Management	12
Information Services	11
Data Access	10
Streams	10
Events	9
Communication	7
Steering	5
Logical Files	3
Data Bases	1

Table 1: Functional areas covered by the use cases, ranked by occurrences in these use cases.

Non-Functional Requirements: Non-functional requirements are orthogonal to the functional requirements: these represent the actions provided by specific calls in SAGA, non-functional requirements describe the desired

Non-Functional Area	#
Error	15
Security	12
Auditing	7
QoS	6
Asynchronous Operations	4
Bulk Operations	2
Workflow	2
Transactions	0

Table 2: Non-Functional areas covered by the use cases, ranked by occurrences in these use cases.

properties of these actions. For example, a `file-copy` operation is a functional element of the API – being able to perform this operation asynchronous and secure, and being able to monitor the operation's progress, are non-functional properties of the API. Table 2 lists the high level non-functional requirement areas of the SAGA API. Many details of these requirements do not make it to the API level, i.e. they are not manifest in specific function calls or flags, but are properties of SAGA implementations. Other areas require explicit support at the API level. The end user does not *need* to interact with any of the non-functional requirements directly (with the possible exception of the error reporting mechanism), and hence these requirements are expressed as optional extensions to the base API.

Design of the SAGA API

The functional requirements are addressed by providing specific objects and method calls, which provide the required programming models and operations. For example, the —saga::rpc— class supports 'Remote Procedure Calls'. The non-functional requirements are provided by consistent syntactic and semantic elements in the API, which (explicitly or implicitly) modify the behaviour of the functional API part. For example, the SAGA task model provides synchronous and asynchronous versions of the above mentioned RPC calls. Finally, the general requirements are addressed by general design guidelines:
Simplicity: the default form for all Grid operations in SAGA is a simple method call on SAGA objects, such as:

```
saga::directory dir (url);
dir.copy (source, target);
```

Additional semantic details (such as asynchronicity of operation, security credentials for operations, notifications on completion etc) are defined as deviations from this standard form. For example:

```
// specify security context for operation (session):
saga::directory dir (session, url);

// specify semantic deviation via flag
dir.copy (source, target, saga::file::Overwrite);

// tag operation to be asynchronous:
saga::task t = dir.copy <Async> (source, target);
```

That way the basic API remains simple, without limiting it to an unacceptably small number of use cases. The syntactic and semantic additions to the basic functional method calls are encapsulated in a set of classes and interfaces used by the functional part of the API; these features correspond to those parts of the API for which there isn't a direct mapping to a function.

The SAGA interface is driven by and implements both functional non-functional requirements

Extensibility: The functional part of the SAGA API is organized in packages. For example, the API offers a 'job package' for job submission and control, a 'file package' for file management and access, a 'stream package' for streaming data between applications, etc. That approach has a number of advantages:

- the API is extensible, as packages can be added rather easily; additionally the functional packages share a common model for non-functional requirements, and these are represented using similar syntax and semantics;
- the user can focus on individual packages, as needed, and can ignore others (Modularity);
- implementations can provide packages separately, lowering the initial effort for implementing SAGA;

SAGA packages implement the functional requirements of the SAGA API.

Figure 1 gives an overview of the SAGA Core API. Although details may not be discernible, the overall structure should be confirmed: non-functional interfaces and classes (top) are used by a number of small and focused API packages (bottom). The packages provide high level abstractions such as streams, files, jobs etc.

The different requirements sets are essentially orthogonal, in the sense that the general, functional, and non-functional requirements can all be approached independently from each other and do not influence each others the design features.

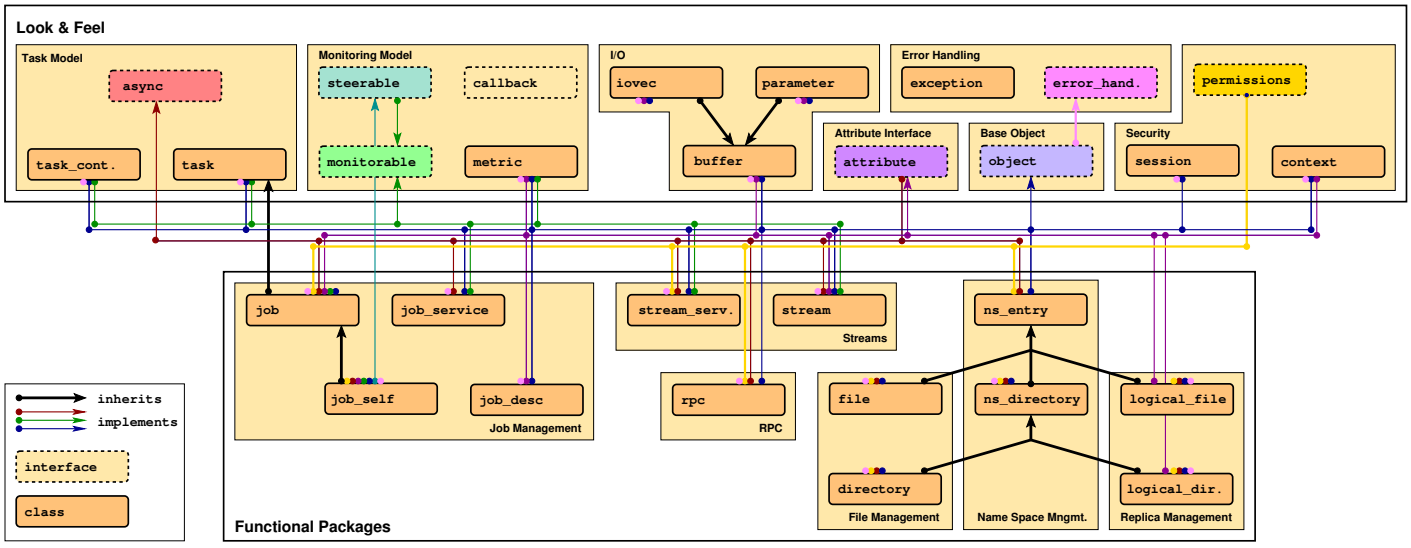

Figure 1: SAGA class hierarchy

SAGA: The Emerging Standard for Application Level Grid Functionality

SAGA's goals – simplified application development and independence from evolving Grid middleware – are impossible without (a) acceptance in the user community, and (b) broad support by various middleware distribution. Middleware vendors and developers are less likely to support such an interface without the conviction that there is a broad-based requirement and the assurance of a wider user community. On the other hand, applications developers are less likely to use an interface that is not commonly available or supported – not to mention that they are less likely to be able to deploy over different platforms.

To address both problems concurrently, and to solve that circular dependency, SAGA has *ab initio* been developed and proposed as an emerging specification of the Open Grid Forum (OGF). The standardization process for the SAGA API is heavily dominated by the same design guidelines as listed in the earlier sections (top-down approach, 80:20 rule, extensible and modular, high level abstractions, distinction of functional and non-functional properties). The SAGA Research Group at the OGF is responsible for the evaluation of use cases and proposing extensions to the core API; the SAGA Core Working Group within OGF is responsible for defining the SAGA API which implements these requirements. This procedure ensures the top down approach and the 80:20 rule is maintained as well as future extensions continue to be use-case driven.

The SAGA Core Working Group made a very conscious effort to reflect the concept of having separate sets of functional versus non-function requirements in the API design. Both are addressed in the main API specification document [**?**], whereas the first half of that document addresses the non-functional packages (URLs, error handling, monitoring, notification, asynchronous operations, memory management security, and generic attributes), the second half addresses a number of functional packages (jobs, files, replicas, streams, and remote procedure calls). The specification is thereby kept language neutral (it uses Scientific IDL, SIDL). The structure allows and encourages add additional API specification documents, which are then to address additional functional API aspects.

A number of API extensions being prepared at the time of this writing, most notably (with their planned OGF document number):

- Service Discovery API package (allows to discover Grid services; GFD.91)
- Message Bus API package (provides a wide variety of communication patterns; GFD.94)
- Advert API package (provides the ability to store application level information persistently, and to publish SAGA objects; GFD.95)
- Checkpoint and Recovery API package (provides a uniform API for *application level* checkpoint creation and management; GFD.96)
- Database Access and Integration API package (provides an interface to the OGF standard DAIS)
- possibly a more advanced interface for job creation and management, in sync with the evolution of the DRMAA, BES and JSDL standards in OGF.

The SAGA Core API are designed to be compatible with the majority of OGF specifications/proposed recommendations [**?**] – BES, GridFTP, JSDL, DRMAA, GridRPC, RNS, and ByteIO[1] – which means that the SAGA API and any SAGA implementations are designed to be compatbile and implementable with middleware which includes, but is not limited to those which implement OGF standards.

The SAGA Core Working Group is in the process of standardizing various language bindings for the (language neutral) SAGA API, as well as for its current and future extension packages. This work is currently focused around Java and C++, as they are the most demanded languages at the moment, but will eventually encompass other languages such as C, FORTRAN and Python. The landscape of SAGA specification documents is depicted in Figure 2, along with the relation of specifications and implementations.

SAGA Reference Implementations

The OGF [**?**] standardization process requires two independent reference implementations to be available before a proposed specification will be accepted as a final standard. At the time of writing there are two distinct groups working on independent implementations of the proposed SAGA specification: a group at the Vrije University (Amsterdam, Netherlands) is working towards an implementation of the SAGA specification in Java. A second group at the Center of Computation and Technology at the Louisiana State University (Baton Rouge, USA) is working on a C++ implementation (along with C, Python, and FORTRAN versions of the specifications) [**?**]. As the SAGA API is originally specified using the Scientific Interface Description Language (SIDL [**?**]), these implementation also represent efforts to develop the corresponding SAGA language bindings specifications.

[1] BES - service interface GFD.108; GridFTP - protocol, GFD.51, GFD.20; JSDL - language GFD.65, GFD.111 and GFD.115; DRMAA - API GFD.22; GridRPC - API, GFD.52; RNS - service interface, GFD.101; ByteIO - service interface, GFD.87, GFD.88

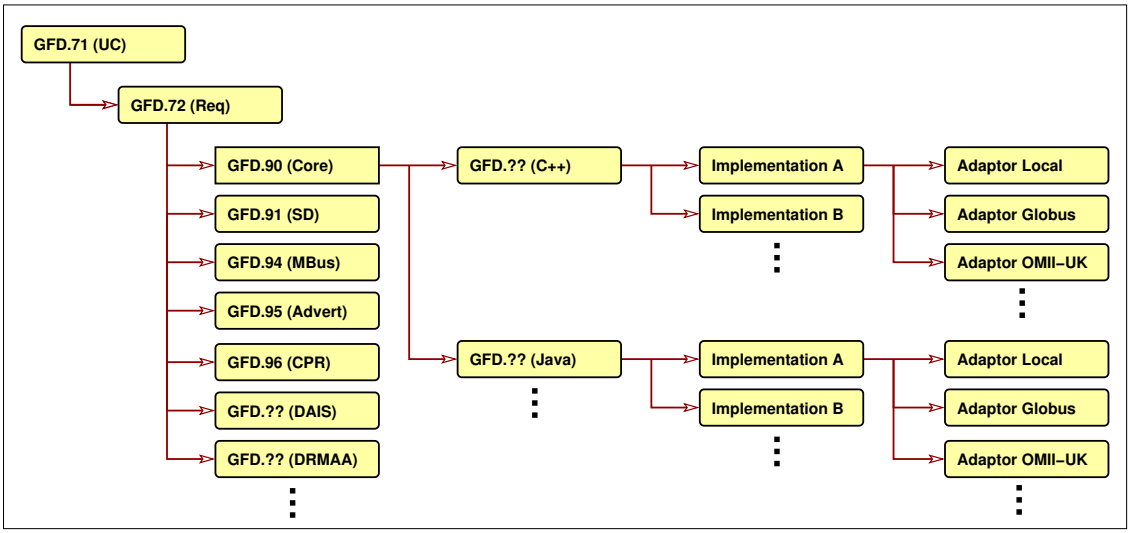

Figure 2: SAGA standard landscape.

Any implementation of the SAGA specification must cater to the development and deployment requirements of applications in a grid environment. They must provide a "simple" and "easy-to-use" API – a primary original motivation for SAGA. Additional design objectives of any successful implementation include, but are not limited to:

- It must cope with 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 evolving middleware, and should allow various incarnations of grid middleware to co-exist.
- It must actively support fail safety mechanisms, and hide the dynamic nature of resource availability.
- It must be portable and platform independent - both syntactically and semantically.
- It must allow latency hiding techniques to be implemented.
- 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.

The C++ implementations (see figure **??**), has the following key features, which support the general requirements of any SAGA implementation listed above:

- Extensible to new adaptors and functional requirements. API extensions are greatly simplified by a generic call routing mechanism, and by macros resembling (SIDL) used in the SAGA specification.
- Modular in design to support flexibility and extensibility. A modular architecture also minimizes the run-time memory footprint.
- Synchronous, asynchronous and task (same as asynchronous, but execution has not been started so far) oriented versions of every operation are transparently provided.
- Dynamically loaded adaptors bind the API to the appropriate grid middleware environment, at run-time, link time binding is also supported.
- Adaptors are selected on a call-by-call basis (late binding, supported by an object state repository), which allows for incomplete adaptors and inherent fail safety.
- Latency hiding (e.g. asynchronous operations and bulk optimizations) is generically and transparently applied.
- Portable across heterogeneous platforms

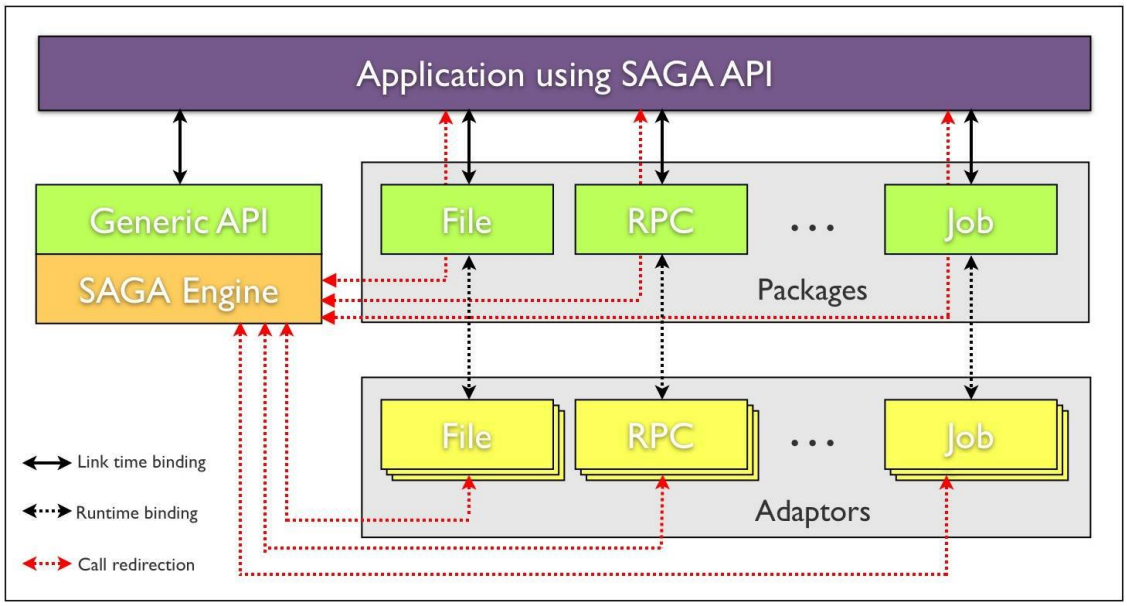

Figure 3: Architecture: A lightweight engine dispatches SAGA calls to dynamically loaded middleware adaptors.

From an end-user perspective, the SAGA reference implementations provide the following:

Consistency Across Programming Languages: The SAGA API specification uses SIDL which is language independent, with language specific bindings in the most common languages. Specific language (Java and C++) implementations have the similar semantics as the SIDL specification and follows the SAGA API specification closely, as will subsequent language bindings to these implementations.

Support for Diverse Middleware and Dynamic Environments: The need for any API implementation to operate over a broad range of middleware distributions requires an adaptor mechanism - which become the only middleware distribution specific component. Late binding, fall back mechanisms, and flexible adaptor selection allow for additional resilience and operation in a dynamic run time environment.

Modularity ensures Extensibility: Implementations are able to cope with the expected evolution and extension of the SAGA API. The adaptor mechanism allows for easy extensions of the library as well as providing additional middleware bindings.

Portability and Scalability: Heterogeneous distributed systems require portable code. The implementations are carefully written to be very portable, being developed on Windows, Linux and MacOS concurrently.

Simplicity for the End User: SAGA is *designed* to be simple. However, simplicity of an API is not only determined by the specification, but also by its implementation, such as the simple deployment and configuration, resilience against lower level failures, adaptability to diverse environments, stability, correctness, and peaceful coexistence with other tools and libraries.