SAGA: A Simple API for Grid Applications
High-Level Application Programming on the Grid

Tom Goodale*†, Shantenu Jha‡, Hartmut Kaiser†, Thilo Kielmann§, Pascal Kleijer¶,
Gregor von Laszewski‖, Craig Lee**, Andre Merzky§, Hrabri Rajic††, John Shalf‡‡

*Cardiff University, Cardiff, UK
†Louisiana State University, Baton Rouge, USA
‡University College, London, UK
§Vrije Universiteit, Amsterdam, The Netherlands
¶NEC, HPC Marketing Promotion Division, Japan
‖Argonne National Laboratory, Chicago, USA
**The Aerospace Corporation, USA
††Intel Americas Inc., USA
‡‡Lawrence Berkeley National Laboratory, Berkeley, USA

http://wiki.cct.lsu.edu/saga/

***Abstract*— Grid technology has matured considerably over the past few years. Progress in both implementation and standardization is reaching a level of robustness that enables production quality deployments of grid services in the academic research community with heightened interest and early adoption in the industrial community. Despite this progress, grid applications are far from ubiquitous, and new applications require an enormous amount of programming effort just to see first light. A key impediment to accelerated deployment of grid applications is the scarcity of high-level application programming abstractions that bridge the gap between existing grid middleware and application-level needs.**

The Simple API for Grid Applications (SAGA [1]) is a GGF standardization effort that addresses this particular gap by providing a simple, stable, and uniform programming interface that integrates the most common grid programming abstractions. These most common abstractions abstractions were identified through the analysis of several existing and emerging grid applications. In this article, we present the SAGA effort, describe its relationship to other grid API efforts within the GGF community, and introduce the first draft of the API using some application programming examples.

***Index Terms*— Grid Programming, API, GGF, Grid Applications, SAGA**

I. Introduction

Many application developers wish to make use of the exciting possibilities opened up by the advent of the Grid. Grid APIs have traditionally been developed using a bottom-up approach that exposes the broadest possible functionality, but requires extremely verbose code to implement even the simplest of capabilities. Applications developers, however, are typically focused on the high-level capabilities to be delivered by their end-user applications, and are daunted by the complexity of the vast array of grid technologies and APIs that currently exist.

SAGA complements these existing grid technologies by applying an application-oriented, top-down approach to programming interfaces that presents the developer with the higher-level programming paradigms and interfaces that they are used to. Such high-level interfaces provide a more concise syntax for expressing the goals that application programmers typically have in mind such as moving a file from point ”A” to point ”B”, or POSIX-like I/O on remote files. They are also intended to cover the most common grid programming operations, such as file transfer, job management, and security.

For example, the complete process of copying of a remote file may involve interaction with a replica location service, a data transport service, a resource management service, and a security service. It may involve communication via LDAP/LDIF, GRAM, HTTP, and GSI, as protocols or protocol extensions. All a C or Fortran application programmer wants to see, however, is a call very much like:

```
call fileCopy (source, destination)
```

Although the file copy example above is simplified, it illustrates the central motivation for our work. SAGA facilitates rapid prototyping of new grid applications by allowing developers a means to concisely state very complex goals using a minimum amount of code. As such, SAGA frees developers to focus their effort on components that require more sophistication than a simple API allows rather than attempting to assimilate the full depth and complexity of low-level grid interfaces all at once. Over time, developers can tune their applications by incrementally replacing the SAGA interfaces by programming directly to the underlying grid interfaces.

It seems obvious, that a complete coverage of all the capabilities offered by a complex grid environment cannot possibly be delivered by a deliberately simple API. Indeed, the single main governing design principle for the SAGA API is the 80:20-Rule: “Design an API with 20% effort, and serve 80% of the application use cases”. So, the SAGA effort intentionally does *not* try to cover all use cases, but only those which are most useful to a large portion of application use cases.

Many projects, such as GridLab [2] with its Grid Application Toolkit [3] (GAT), the Commodity Grid project (CoG) [4], and RealityGrid [5] recognized the need for higher-level programming abstractions and have sought to simplify the use of the grid for application developers. Simultaneously, many application groups in fields ranging from particle physics

to computational biology, have developed their own APIs to simplify access to the grid for their application such as the Particle Physics Data Grid [6] and the Illinois BioGrid [7]. This momentum has culminated in the formation of the Simple API for Grid Applications Research Group (SAGA-RG) within the Global Grid Forum (GGF).

SAGA was founded at GGF11 in Hawaii, after successful BoFs at GGF10 and GGF9 and originally chaired by members of the GridLab, Globus/Python-CoG, and RealityGrid project. The aim of the group has been to identify a set of basic grid operations and derive a simple, consistent API, which eases the development of applications that make use of grid technologies. The implementation of the SAGA API has been guided by examination of the requirements expressed by existing and emerging grid applications in order to find common themes and motifs that can be reused across more than one use-case. The broad array of use cases that have been collected from the nascent grid application development community continues to play a pivotal role in all design decisions.

The next section describe the process which has been used to derive the initial scope of the API in more detail, and also documents general design principles applied to the initial implementation. The SAGA API is then put in relation to other API related work in GGF, and also to projects with similar scope, such as the GAT and Globus-CoG. A more detailed description of the current version of the SAGA API specification follows, including a number of explicit code examples.

II. SAGA: General Design

The original SAGA Research Group charter outlined a systematic approach to deriving the API: use case analysis, requirements analysis, and API development. API development was further split into the creation of an abstract, language independent, specification, and then a set of language bindings covering the major languages used in scientific application development — Fortran, C, C++, Java, Perl and Python. At GGF12 in Brussels, it was suggested that a design team should be formed to rapidly proceed through the analysis process and produce a strawman API that could be used for further discussion and refinement into a final API. That SAGA Strawman API was supposed to be iterated against the use case requirement analysis which was to be produced in parallel, in order to stay true to the process outlined in the charter. [1]

A. Use Case Analysis

Upon creation, the SAGA group sent out a call for use cases, which resulted in 24 responses. These came from a wide range of applications areas, such as visualization, climate simulations, generic Grid job submission, application monitoring and others. A large set of use cases (9) was submitted by the GridRPC Working Group, which targets a similar set of application in GGF (see section III-A).

Common to the majority of the submitted use cases was their explicit focus on scientific applications (not surprising, as almost all use cases came from the academic community). For a discussion of non-scientific applications as SAGA customers, see subsection II-C.

The analysis of the SAGA use cases focused on the identification of the SAGA API scope, on the level of abstraction needed and wanted by the application programmers, and on non-functional requirements to the API, such as targeted programming languages, requirements to thread safety etc. Additional similar requirements are also drawn from other prominent projects (such as listed in section IV), which all took active participation in that SAGA group effort.

B. The Design Team and the SAGA Strawman API

In response to the suggestions at GGF12, the SAGA group sent out a call and formed a design team, which met in December 2003 under the auspices of the Center for Computation and Technology, at Louisiana State University, Baton Rouge, and at the Albert-Einstein-Institut, Golm, Germany, with an AccessGrid link connecting the two halves of the design team, and allowing occasional outside participation. The major participants were Tom Goodale, David Konerding, Shantenu Jha, Andre Merzky, John Shalf, and Chris Smith, with Craig Lee, Gregor von Laszewski, and Hrabri Rajic providing important input. This was followed in January of 2004 by a meeting held at the Lawrence Berkeley Lab in Berkeley, California, USA, also with an AG-Link to the Vrije Universiteit, Amsterdam, Netherlands.

Before embarking on developing the APIs, a few general design issues were considered and agreed upon.

- The API would be developed and specified in an object-oriented manner, using the Scientific Interface Description Language (SIDL) to provide a language-neutral representation. The consensus was that it would be easier to provide a mapping from an object-oriented representation into a procedural language such as Fortran or C, than to provide a mapping from a procedural representation into an object-oriented language such as C++ or Java.
- Any asynchronicity would be handled by a polling mechanism rather than a subscribe/listen mechanism. This makes the implementation in non-multi-threaded environments clearer, and avoids defining some mechanism to represent closures in non object-oriented languages. While both these things are possible — co-operative multi-tasking using a call to a routine giving time to asynchronous tasks, and closures by use of a data-pointer, function-pointer pair — they would provide greater divergence between the different implementation languages. The team did not rule out future versions of the API from having such features[2].
- Each grid subsystem should be specified independently from other subsystems, and should not make use of them as far as possible. This allows the independent development and implementation of parts of the API, at the cost of a small extra amount of complexity when using data derived from one part of the API in another,

[1] Currently, the SAGA group is undergoing a re-chartering process and transforms into a Working Group, and is here referred to as SAGA-WG.

[2] But see notes to asynchronous notification below.

such as when opening a logical file for remote file access, which requires the retrieval of the location of the logical file from the replica catalog, using one system, and the using the File APIs to open it. The team decided that the extra complication did not outweigh the advantages gained by the semantic simplicity of defining things separately, and the ability to to implement each subsystem independently of other subsystems. It is possible that future versions of the API may have bridging calls to simplify some uses of the APIs, however these would be built upon the underlying subsystems, thus not breaking their independence.

The design team examined the use cases and extracted the key areas which were crucial to a number of the use cases:

- *Sessions and Security.* Security is essential for any grid applications that run across multiple administrative domains or security boundaries. In SAGA, security is managed with the notion of a *session*.
- *Data Management.* This can take many forms but *remote file access* is a common requirement for many grid applications. The team also tackled replica catalogs [8], as it was decided that the technology was sufficiently mature and the interfaces were sufficiently similar to files to make this reasonable.
- *Job Management.* The concept of *remote job startup* is a very common requirement. *Asynchronous operations* were also found to be an important paradigm for remote grid operations and, hence, are represented as *tasks* in the initial SAGA Strawman API.
- *Inter-process Communication.* SAGA supports inter-process communication with a *stream* concept that is similar to BSD sockets.

These key areas are discussed in detail in Section V.

As the SAGA API requirement definition work are performed in parallel to the SAGA Strawman API definition work (see below), the SAGA group faces the additional task to iterate the SAGA Strawman API against these requirements. That process is currently underway, and seems to yield promising results: those use cases already evaluated seem to be easily implementable with the current API definition, as long as the scope matches. However, that scope match is definitely incomplete. In particular, several use cases miss the ability of SAGA to provide *asynchronous notification*. That feature is therefore planned to be integrated into the API draft soon.

C. SAGA and Non-Scientific Applications

As has been discussed in the previous subsection II-A, it is possible that the SAGA Strawman API reflects a perceived bias towards scientific application. SAGA requirements, goals and design are use-case driven. Given that till the time of writing of this report, the SAGA-WG had received use cases from only the scientific community, it is natural that SAGA does reflect this bias – it is not intentional but an artifact of the set of use cases received. Indeed, SAGA does not by definition constrain itself to scientific applications only. It is undeniable that the need for well-designed APIs goes well beyond scientific computing. The recent, very popular phenomenon of Google mashing is possible due to well designed and simple APIs that are openly publicized by Google [9] – thus enabling mash-up application developers to use Googles' capabilities without ever having to worry about the complex and in this case proprietary underlying technology and implementation. In many ways, this is precisely the philosophical aim of a Simple API for Grid Applications. Admittedly, the API for Google mash-ups isn't strictly a grid API, but the argument in favor of simple API for grid applications is still valid. What is, however, unclear at the time of writing is, if the same "Simple" API will be equally effective in both domains of scientific computation and commercial grid applications. As part of the exercise of determining that any eventual Simple API is of use in both areas as well as the initial first steps in this direction – direct and indirect – the SAGA-WG has, i) asked other relevant groups at the GGF for further "non-scientific computing" use-cases, and ii) agreed to work more closely to ensure compatibility with groups involved in developing relevant middleware, such as OGSA [10]. As these middleware groups have in turn received input from both communities, SAGA will hopefully benefit indirectly.

III. SAGA IN GGF

SAGA has ties to a number of other groups in GGF. In fact, a significant part of the SAGA input and motivation originates in the work of other GGF groups. This sections lists the most important of these groups, and shortly describes their relation to the SAGA-WG.

A. DRMAA-WG

The DRMAA Working Group [11] at GGF produced a API specification for the submission and control of jobs to one or more Distributed Resource Management (DRM) systems. The scope of this specification is all the high level functionality which is necessary for an application to consign a job to a DRM system, including common operations on jobs like termination or suspension. The objective is to facilitate the direct interfacing of applications to today's DRM systems by application's builders, portal builders, and Independent Software Vendors (ISVs) [12].

DRMAA and SAGA groups have enjoyed several years of active collaboration. While the motifs of the SAGA people were not to reinvent the wheel (i.e. the DRMAA API), the DRMAA people had an interest in independent scrutiny of its API efforts, in particular for job submission and job states. Both groups have a goal of a common base API which provides an easy transition for application developers moving from DRM systems or traditional Compute centers to the Grid systems or vice versa. DRMAA has passed its work experience during the SAGA Strawman API deliberations and has participated in many SAGA group initiatives, while at the same time attracting SAGA principals' participation in its sessions. Currently, the job submission part of the SAGA API draft reflects the DRMAA API very closely.

B. GridRPC-WG

The GridRPC API [13] represents ongoing work to standardize and implement a portable and simple remote procedure call (RPC) mechanism for grid computing. This standardization effort is being pursued through GridRPC Working Group at GGF. The initial work in that group showed that client access to existing grid computing systems such as NetSolve [14], Ninf [15], and DIET [16], can be unified via a common API, a task that has proven to be problematic in the past. The group successfully created such an API specification, which is now a GGF Draft Recommendation [17].

The cooperation with the SAGA-WG allows the GridRPC-WG to broaden the target user community for GridRPC significantly, and would also equip the GridRPC API with a look & feel which is compatible with the one used for other grid programming paradigms. For example, the model for asynchronous calls used for RPCs will match the model for asynchronous calls of other paradigms, e.g. remote file access.

C. GridCPR-WG

GGFs Grid Checkpoint and Recovery (GridCPR) Working Group [18] is devising an architecture for checkpointing long running applications for later recovery, possibly on different grid resources. GridCPR use cases include fault resilience, intermittent operation (using small time slices until final completion), as well as parameter space exploration using various, related checkpoints.

The currently envisioned architecture encompasses services for authentication/authorization, job management, checkpoint state management, checkpoint transfer, and event handling. Applications are supposed to use a GridCPR library, providing an API to these services. Details of this API are currently under investigation. As several concepts are similar to those defined by the SAGA API (copy/move checkpoint files, read/write data, find files etc.), compatibility with SAGA is desired. That would provide application programmers with a more complete and consistent interface to the grid related capabilities offered by the APIs.

D. OGSA-WG

The Open Grid Services Architecture (OGSA) Working Group at GGF defines a Web Service based architecture which eventually will be the overarching design principle for all grid services. The problem space targeted by the OGSA group at first seems unrelated to the SAGA effort but, in fact, represents the class of service-oriented, middleware architectures that could be the target platform for many SAGA implementations. Such service-oriented architectures address many issues, such as resource naming, state management, and security, that will require a simplified mapping into the SAGA API. Hence, the SAGA-WG is in active communication with the OGSA group and other OGSA-related efforts to ensure the proper level of compatibility by cross-reviewing their respective standard documents and by providing SAGA reference implementations on OGSA-based middleware systems.

E. Other GGF Working Groups

The Grid File Systems (GFS) Working Group [19] targets the definition of a architecture which integrates physical file systems into a single global, uniform name space [20]. The SAGA-WG uses the definitions for name spaces defined by the GFS-WG, and hence has the same notion of files and directories as that group.

The Byte-IO Working Group [21] defines an OGSA compatible service which allows for efficient remote access to binary data, e.g. files. Also, the GridFTP Working Group [22] defined extensions to the standard FTP protocol which allows for efficient memory-to-memory and disk-to-disk data transfer in grids. Both groups invest significant time in exploring paradigms for efficient remote data access. These paradigms are partially reflected by the SAGA API.

The JSDL Working Group [23] defines a XML based Job Submission and Description Language. Although the SAGA API currently refrains from any explicit usage of XML, the keywords used for the SAGA job descriptions reflect those defined by JSDL.

The Persistent Archive (PA) Research Group [24], Data Replication (REP) Research Group [8] and the OGSA Replica Services (OREP) Working Group [25] have all worked on the services that manage distributed data files for data grids. Their work [26] is reflected in the SAGA Logical File API, which adopts the concept of logical files (or 'replicas') that was defined by these groups.

IV. Related Work

The work on the SAGA API is not an isolated effort, but builds on top of a number of previous projects targeting the same problem space. The Globus-CoG is probably the most prominent representative of an API which tries to specifically abstract the Globus package. The GAT from the EU-GridLab project targets already a more general abstraction of grid paradigms, and follows similar design principles as SAGA. Other projects such as RealityGrid and DataGrid/EGEE/LHC target the same problem space, but with interfaces with stronger focus on specific community needs. This section describes these efforts and their influence on SAGA in some detail. In general, the SAGA-WG can be seen as the "Grand Unification" of all these grid APIs, which provides a more uniform, middleware independent and standardized solution for grid application programmers.

A. GAT

The design of the Strawman API and the ongoing work on a SAGA engine implementing this Strawman API is very much influenced by the experiences collected with the Grid Application Toolkit (GAT [3]).

Figure 1 shows the GAT architecture. It mainly distinguishes between user space and capability space. In user space runs the application code that has been programmed using the GAT API. The GAT *engine* is a lightweight layer that dispatches GAT API calls to service invocations via GAT *adaptors*. Adaptors are specific to given services and hide all service-specific details from the GAT. A GAT engine typically loads

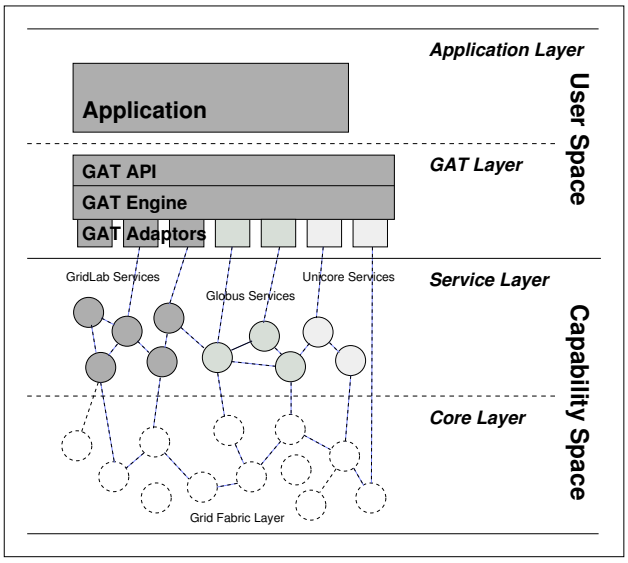

Fig. 1. The GAT framework software architecture.

adaptors dynamically at runtime, whenever a certain service is needed. The GAT functionality is grouped into various, so-called *subsystems* such as

- **The GAT Base Subsystem** which defines general objects and methods, supporting interaction with the GAT engine.
- **The Data Management Subsystem** which covers three areas: interprocess communication, remote file access, and file management.
- **The Resource Management Subsystem** which deals with grid (compute) resources. It provides functions for both resource discovery and job management.
- **The Event and Monitoring Subsystem** which allows applications to send and receive events, such as events generated by a grid monitoring service.
- **The Information Management Subsystem** which connects to a generic grid information system.
- **The Utility Subsystem** which contains a number of classes providing general, convenience, and/or utility-oriented methods.

An engine may load multiple adaptors implementing the same subsystem at the same time. For example, one adaptor can give access to a local file system while another one gives access to files via GridFTP [22], and a third one is using a replica catalog [27].

While application and GAT together run in user space, the services are executed in the capability space, which can be distributed across the machines of a Virtual Organization (VO [28]), including the one running the application. The capability space consists of the resources themselves, and the middleware providing services to access them.

The GAT API has been designed using an object-based approach, trying to avoid dependencies on any specific language binding [29]. However, for each supported application programming language, a complete GAT layer has been provided, consisting of API, engine, and adaptors. Currently the GAT supports C, C++, Python and Java as target languages, where the C, C++ and Python implementations share the same engine and the same set of adaptors (regardless of the language these are written in).

The experience with the GAT showed, that the implemented API was partly too terse and too much oriented to synchronous operation. The SAGA Strawman API tries to overcome this limitation by defining a more complete set of functions dealing with different aspects of grid related interfaces, such as fuller namespace support for file systems and by defining a dual interface to all of the provided functionality, synchronously and asynchronously.

B. Commodity Grids (CoG) Kits

The Java CoG Kit [30] has over the years provided successfully access to grid services through the Java framework. Components providing client and limited server side capabilities are available. A subset of the Java CoG Kit is distributed with the Globus Toolkit versions 3.0.2, 3.2, 3.9 and 4.0 as they use it. The Java CoG Kit significantly enhances the capabilities of the Globus Toolkit by introducing grid workflows, control flows, and a task based programming model. A testing framework is available to conduct client side functionality tests. Portals and Swing components are under development [4]. A Python version of the CoG [31] does also exist, and has a significant user base.

The CoG kits have a very similar target user group as SAGA. Also, as the CoG kits are around and used since a number of years by now, the accumulated feedback and experience gathered by the projects is very valuable to the SAGA group. The group tries to include as much of that knowledge as possible into its API design.

C. RealityGrid

RealityGrid is a UK e-Science pilot project that has used grid-based computational steering for a wide range of scientific applications Computational steering as used by RealityGrid is defined as any mechanism that enables the control of the evolution of a computation, based upon the real time analysis and visualization of the status of a simulation, as opposed to the post facto analysis and visualization of the output of a computation. The functionality referred to as computational steering enables the user to influence the otherwise sequential simulation and analysis phases by merging and interspacing them. The innovative use of computational steering and interactive simulations on a grid of high-end resources has shown to be effective in solving scientific problems [32], [33], which probably couldn't be otherwise. The use of grid-based computational steering for scientific problems has amongst things raised a need for user-level control and thus APIs related to advanced reservations, co-allocation of resources and the interaction of grid-middleware with optical control-planes. Providing for all of these needs is well outside of the scope of initial efforts of SAGA Strawman API, but a SAGA style look-and-feel API which does address these requirements will eventually be required by the scientific grid-computational scientist.

D. gLite and others

A number of more domain-specific interfaces exist which try to shield application developers from the complexities and dynamic behavior of grid middleware in that domain. Notably amongst these interfaces is gLite's 'Grid Access Service' (GAS) [34]. Similar to SAGA, the GAS defines a Service which provides a simplified, stable and abstract interface to the complete gLite middleware infrastructure. Despite its differing architectural approach, the goals and scope of the GAS is very similar to the SAGA effort, and are considered as valuable input to the SAGA design.

V. SAGA DETAILS

As of October 2005, the SAGA API specification is converging toward a coherent document. The intended scope seems well covered, and the use cases which have been motivating that scope seem to be well implementable with the current API version. SAGA endeavors to provide the grid applications programmer with a common look & feel across all functional areas. Any potential future extensions of the SAGA API will inherit the same look & feel and so form an integral and easy to learn new parts of the API.

This section describes the API in its current state in some detail, and shows examples of its usage. However, this description and the examples are not meant to be normative, and will with some certainty differ in many details from the final SAGA API specification, which is expected in 2006.

A. Session Handling and Security

An essential part of the SAGA programming model is formed by the API's session management and security handling. In order to keep the API usage simple and transparent to the application, however, any session and security handling is optional. A default session handle is used if no explicit handle is specified, and a SAGA implementation is required to initialize available default security tokens as available, e.g., the default Globus certificate is to be used if it is found in its standard location.

However, if required, an application can create multiple independent session handles. To each of these sessions, explicit security tokens (called `context` in SAGA) can be attached. Any SAGA object creation and method invocation can then associated with that session handle, and is required to use the specified security tokens. The code example in Figure 2 shows how to access a remote file using a myproxy login.

Fig. 2. Code Example: *Session and Security Handling in SAGA*

```
#include <iostream>
#include <saga.h>

void sample_session ()
{
  try
  {
    // create a SAGA session handle
    saga::session s;

    // create a myproxy context ...
    saga::context c (saga::context::MyProxy);

    // ... and fill it
    c.set_attribute ("username", "andre");
    c.set_attribute ("password", "secret");

    // attach the context to the session
    s.add_context (c);

    // create a file in that session
    saga::file f (session,
        "gsiftp://ftp.host.net/pub/INDEX");

    // further code see later file examples
  }
  catch ( saga::exception e )
  {
    std::cerr << e.get_message () << std::endl;
  }
}
```

B. Data Management

Remote data access and management certainly belong to the most understood and most widely used paradigms in distributed environments. Grids however apply subtle changes to the well known remote paradigms, which need to be reflected in the SAGA API. The data management parts of the API are hence a blend of well known paradigms (such as POSIX like directory navigation) and new grid paradigms (such as replica management).

1) Namespaces: Many data oriented services offer hierarchical categorization for the respective data entities, and support the notion of containers or directories. That paradigm is e.g. used for file systems, replica catalogs, grid information systems etc. Also, URIs as the mostly used mechanism for resource addressing can be mapped to hierarchical name spaces. In order to provide a uniform representation of these various name spaces, the SAGA API includes an abstract interface for (`NameSpace`), which is to be implemented by the various data management subsystems (e.g. for files and logical files). That name space interface covers the management of the respective data elements as entities, and provides methods to navigate in the name space, to create, copy, move and delete elements in the name space, and to query the elements for information (such as name and type). The code example in Figure 3 shows how to use that interface to create a directory on a remote ftp server.

2) Physical Files and Directories: The SAGA API defines the management of physical files in the paradigm of name spaces as described above. Hence the SAGA representations for remote files and directories allow to navigate remote file systems, and to manage the remote file system entities, i.e. files and directories. Additionally to the name space interface, the SAGA file object allows to access the *content* of the remote files, as it provides *read*, *write* and *seek* methods. The example in Figure 4 illustrates the usage of a remote file read.

If performance is an issue for the application, remote file access can become a tricky business. Various mechanisms are known in the distributed computing community to handle that issue, however, there is currently no single known solution

Fig. 3. Code Example: *Create a remote directory*

```
#include <iostream>
#include <saga.h>

void sample_mkdir ()
{
  try
  {
    // open a remote directory
    saga::directory d
         ("gsiftp://ftp.host.net/pub/");

    // create a new subdirectory
    d.mkdir ("data/");
  }
  catch ( saga::exception e )
  {
    std::cerr << e.get_message() << std::endl;
  }
}
```

Fig. 4. Code Example: *Read a remote file and copy to stdout*

```
#include <iostream>
#include <string>

#include <saga.h>

void sample_fileread ()
{
  try
  {
    // open the remote file
    saga::file f
        ("gsiftp://ftp.host.net/pub/INDEX");

    // read as long as there are data to read
    std::string s = f.read (100);
    while ( !s.empty() )
    {
      std::cout << s;
      s = f.read (100);
    }
  }
  catch ( saga::exception e )
  {
    std::cerr << e.get_message() << std::endl;
  }
}
```

for all potential file access patterns. The SAGA API hence includes three different paradigms to allow the implementation of various performance mechanisms. These paradigms differ in their respective semantic and syntactic level of abstraction. For details, see the "File" section of the SAGA API specification [35].

3) Logical Files and Replica Catalogs: A data management paradigm newly introduced in grids is that of file replica management. Logical filenames are used to identify a file – that file however can have multiple identical instances distributed over the grid (replicas). A Replica Catalog is keeping track of these instances, and maps the logical file name to physical file names if required.

The SAGA API provides access to that paradigm. In SAGA, the logical file names form again a name space as described above. The `logical_file` object offers additional methods to manipulate the set of instances associated with that logical file name: *add_location*, *remove_location*, *list_locations* and *replicate*. The last method triggers a copy of any instance of the logical file to a new location, while adding that new location to the list of associated instances. That method is shown in the code example in Figure 5

Fig. 5. Code Example: *Replicate a logical file to localhost*

```
#include <iostream>
#include <saga.h>

void sample_filereplication ()
{
  try {
    saga::logical_file lf
           ("rls://rls.host.edu/data/test");

    lf.replicate ("https://localhost/tmp/test");
  }
  catch ( saga::exception e )
  {
    std::cerr << e.get_message () << std::endl;
  }
}
```

C. Job and Task Management

Remote job execution and management naturally plays a very important role in grids – as grids are designed to simplify the virtualization and utilization of remote resources, remote job management is probably the prime usage area at the current stage. The SAGA API tries to greatly simplify the standard paradigms in that area: simple job submission, and retrieval of the jobs output data.

As all distributed systems, grids suffer from potentially large latencies for remote interactions. Hence, for many grid applications, it is often mandatory to manage such interactions asynchronously. For this reason, SAGA introduces a *Task model* for asynchronously managing remote jobs. Highly relevant to this model is the notion of *asynchronous notification*.

1) Jobs: A number of GGF working groups are targeting the various technical details for job and resource management in grids. Some of them do also target grid applications, and specify job description languages, APIs etc. The SAGA API tries to avoid to "re-inventing the wheel", and closely coordinates with these groups. The SAGA job submission API follows closely the Distributed Resource Management Application API (DRMAA), the SAGA job description is mostly derived from the Job Specification and Description Language (JSDL). The SAGA job states do also follow the DRMAA specification. More details on the relation between SAGA and the various other GGF working groups can be found in section III.

The SAGA API offers programmers two noteworthy paradigms: that of a job service, which represents a job submission resource, and that of a remote job. A job is defined by its job description, and is instantiated by submitting that job description to a job service. An instantiated job allows various control and query methods to be called, such as *suspend*, *resume*, *signal*, *migrate* and *terminate*. The job instance can

also be queried for its current state, error conditions etc. Input and output data files and streams are, if required, defined in the job description.

In order to further simplify the job submission, a job service also offers a *job_run* method, which requires only the essential job information to be given, and creates the job description internally. The call returns the standard input, output and error streams for the remotely run job. The code example in Figure 6 illustrated the usage of the *job_run* method in Perl.

Fig. 6. Code Example: *Start a remote /bin/date, and retrieve output (just for a change, that example is given in Perl)*

```
#!/usr/bin/perl -w

BEGIN {
  use strict;
  use saga;
}

eval {
  # create job service
  my  $js = new saga::job_service ();

  # and submit a job
  my ($job, IN, OUT, ERR)
          = $js.runJob ("ssh.university.edu",
                        "/bin/date");

  # print any job output
  while ( $job.getJobState () !~ /^Done/io )
  {
    print STDOUT <OUT>;
    print STDERR <ERR>;
  }

  # status is 'Done' here
};

# catch all saga exception
if ( $@ =~ /^saga/io )
{
  die "saga error: $@\n";
}
```

2) Tasks: To manage the potentially large latencies in grid environments, SAGA introduces an asynchronous Task model. For each synchronous method call (e.g. `file.read (target)`), that model introduces a asynchronous counterpart (`file.asynch.copy (target)`). That asynchronous method call returns a `saga::task` object, which offers methods to watch the state and progress of the operation, and to cancel it if required. The code example in Figure 7 shows the SAGA task model applied to a remote `mkdir` operation[3].

The SAGA Task Model does also include a container class for tasks, which allows to handle large numbers of task objects efficiently (e.g. wait for all of them to complete). There are considerations to extend that task model by allowing to have dependencies between tasks, and so to effectively introduce workflow, but we consider such extensions to be out of scope for SAGA for now.

[3]It must be noted that the SAGA Task Model is currently under discussion, and might change to some extend (without loss of functionality). So the example given here should be considered very preliminary.

Fig. 7. Code Example: *asynchronous version of the remote mkdir example: This example should be compared to the synchronous version in Figure 3.*

```
#include <iostream>
#include <saga.h>

void sample_async_mkdir ()
{
  try
  {
    // open a remote directory
    saga::directory d
         ("gsiftp://ftp.host.net/pub/");

    // create a mkdir task
    saga::task t = d.async.mkdir ("data/");

    // showing the task state
    if ( saga::task::Running == t.get_state () )
    {
      std::cout << "running ..." << std::endl;
    }

    // wait forever
    t.wait (-1);

    // mkdir is done by now.
  }
  catch ( saga::exception e )
  {
    std::cerr << e.get_message () << std::endl;
  }
}
```

3) Asynchronous Notification: Tightly connected to the topic of asynchronous method invocation is the topic of asynchronous notification for events, such es e.g. job status changes. The current SAGA specification abstains from the topic, as it was initially felt that asynchronous notification might be difficult to design for a language agnostic API; however, the group got very specific requirements to add simple callbacks to the API. So work is currently underway to add a callback mechanism to the SAGA Task Model, to allow for asynchronous notification on task status changes and error conditions.

D. Inter-Process Communication

The SAGA API provides only limited support for inter process communication. That problem space is well supported by other, acknowledged systems. For example, versions of MPI exist which allow for tightly coupled communication between distributed application instances in cluster-like setups. Secured SOAP via HTTP(S) is a standard way to communicate with services. SAGA does provide a grid-equivalent of BSD-Sockets. The `saga_stream` is a bi-directional communicating channel, which transports untyped binary data. The example in Figure 8 shows how a application uses the client side of such a communication channel.

The SAGA-WG considers to extend that support for inter process communication, e.g. by support for typed data, or for messages. However, such extensions are not in the SAGA scope for now, and depend on further explicit application use cases.

Fig. 8. Code Example: *Create a streaming client*

```
#include <saga.hpp>

int main (int argc, char** argv)
{
  // create stream to server contact in argv[1]
  saga::stream s (argv[1]);

  // open the connection
  s.connect ();

  // message to write in argv[2]
  s.write (argv[2]);

  // the stream gets destroyed at and of scope,
  // connection is closed in destructor
}
```

E. Possible SAGA Extensions (Future Work)

As described above, the current scope of the SAGA API specification is intentionally very limited, and drawn from a finite set of use cases. However, the SAGA-WG has received a number of requests to widen the scope of the API, and a number of additional use cases for the respective areas have been submitted. SAGA is intended to follow up with these requests in future versions of the specification. This section allows the reader a glimpse of the SAGA future, and describes these extensions. It must be noted though that all material described here is very preliminary, and represents by no means any part of the current SAGA specification.

1) Monitoring and Steering: The ability to monitor remote applications (and in fact any remote grid resource) is integral part of a number of received use cases. SAGA currently only allows polling of specific monitorables, such as the state of jobs and tasks. Application level monitoring and steering is currently not supported. A number of use cases have been submitted to the group though, which require monitoring and steering, in particular in the context of visualization.

The UK-eScience project 'RealityGrid' [5] designed and implemented a grid monitoring and steering infrastructure [36]. The RealityGrid Steering API has been used across a wide range of different scientific applications — ranging from classical molecular dynamics, lattice-Boltzmann simulations and Monte-Carlo simulations of polymers — providing testimony to its flexibility and breadth. Together with the monitoring capabilities of GAT (which derive from the monitoring package Mercury [37]), the RealityGrid API will probably form the basic input for a future SAGA monitoring and steering API.

2) Remote Procedure Calls: As described in section III, the GridRPC Working Group in GGF defined another application oriented API, which provides Remote Procedure Calls (RPCs) in grid environments. The GridRPC-WG and the SAGA-WG started to actively pursue the integration of GridRPC into SAGA. That would provide a consistent look & feel for both APIs, and provide grid application programmers with a more complete and more consistent set of programming paradigms. An example for the current candidate API extension is shown in Figure 9.

Fig. 9. Code Example: *Potential RPC representation in SAGA*

```
#include <iostream>
#include <list>
#include <saga.h>

void sample_rpc ()
{
  try
  {
    // initialize rpc function handle
    saga::rpc handle
          ("gridrpc://remote.host.net/bin_date");

    // initialize in/out argument stack
    std::list<std::string> stack;

    // call the remote function
    handle.call (stack);

    std::cout << stack.pop_front () << std::endl;
  }
  catch ( saga::exception e )
  {
    std::cerr << e.get_message() << std::endl;
  }
}
```

GridRPC currently uses the notion of a *function handle* to invoke and manage remote procedure calls. A major topic currently under discussion in the GridRPC Working Group is the use of *data handles* to manage the flow of arguments between successive remote calls. Such data handles could possibly be managed as remote files in the SAGA API whose lifetime could be explicitly determined by the user, but could also be automatically determined by the lexical scoping of the data handle.

3) Checkpoint and Recovery: The GridCPR Working Group in GGF is, as also described in section III, defining an API which allows applications to interact with its Grid Checkpointing and Recovery infrastructure. That API would also benefit from the coherent look & feel SAGA provides. Discussions with the GridCPR group have been started, and an future inclusion of the GridCPR API into the scope of SAGA is evaluated.

VI. Conclusions

The advent of the new, exciting, promising, but also complex grid technology left application programmers wanting. The Simple API for Grid Application addresses their need for a simple, stable and standardized program development environment.

The SAGA API is defined in synchronization with all related efforts inside GGF, such as the DRMAA, JSDL, GridCPR, GridRPC and OGSA WGs. With OGSA, we are specifically investigating the mapping of service-oriented architectures into the SAGA API. We are working closely with the known related efforts outside GGF, such as the Grid Application Toolkit (GAT) from the EU GridLab project, the Commodity Grid (CoG) Toolkits from the Globus project, and the RealityGrid application monitoring and steering infrastructure.

The requirements for the SAGA API scope and structure are drawn from a reasonably large set of application use cases.

In order to produce a usable API specification timely (the perceived need for that API is continually increasing), a design team was formed in side the SAGA-WG. That design team drafted the SAGA Strawman API, which is currently under scrutiny to check its suitability for the SAGA use cases. This article has presented the SAGA Strawman API and illustrated the simplicity of its usage with a number of code examples.

The future work of the group is to finalize the API specification, to support the various ongoing reference implementation efforts, and to later expand the scope of the SAGA API to other relevant areas covered by the respective use cases.

ACKNOWLEDGMENTS

The presented work is the result of a very large community effort, and the number of involved people is way too large to list them all individually. That list includes a large number of application developers which provided input to the SAGA Working Group in form of use cases. The input and discussions with the other working groups in GGF has been crucial to the successful design and scoping of the SAGA API. The design team, whose members have already been mentioned, have invested significant time and effort to draft the current SAGA API. All projects listed in the related work section, funded by a large and diverse set of institutions and funding agencies, have been in close contact with the SAGA effort, and actively participated in shaping the SAGA API. The SAGA Working Group wants to thank all of them.

REFERENCES

[1] GGF. (2004) Simple API for Grid Applications Research Group. [Online]. Available: http://forge.gridforum.org/projects/saga-rg/

[2] G. Allen, K. Davis, K. N. Dolkas, N. D. Doulamis, T. Goodale, T. Kielmann, A. Merzky, J. Nabrzyski, J. Pukacki, T. Radke, M. Russell, E. Seidel, J. Shalf, and I. Taylor, "Enabling Applications on the Grid – A GridLab Overview," *International Journal on High Performance Computing Applications*, vol. 17, no. 4, pp. 449–466, 2003.

[3] G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A. Merzky, R. van Nieuwpoort, A. Reinefeld, F. Schintke, T. Schütt, E. Seidel, and B. Ullmer, "The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid," *Proceedings of the IEEE*, vol. 93, no. 8, pp. 534–550, 2005.

[4] G. von Laszewski, I. Foster, J. Gawor, and P. Lane, "A Java Commodity Grid Kit," *Concurrency and Computation: Practice and Experience*, vol. 13, pp. 643–662, 2001. [Online]. Available: http:/www.cogkit.org/

[5] RealityGrid Project. (2005) The RealityGrid Project. [Online]. Available: http://www.realitygrid.org/

[6] Particle Physics Data Grid. [Online]. Available: http://www.ppdg.net

[7] Illinois BioGrid. [Online]. Available: http://gridweb.cti.depaul.edu/twiki/bin/view/IBG/AboutIBG

[8] GGF. (2004) Data Replication Research Group. [Online]. Available: http://forge.gridforum.org/projects/jsdl-wg/

[9] D. Darlin, "A Journey to a Thousand Maps Begin With an Open Code," *New York Times*, Oct. 20th, 2005. [Online]. Available: http://www.nytimes.com/2005/10/20/technology/circuits/20maps.html?emc=%eta1

[10] I. Foster *et al.*, "The Open Grid Services Architecture, Version 1.0," January 2005, http://www.ggf.org/documents/GFD.30.pdf.

[11] GGF. (2003) Distributed Resource Management API Working Group. [Online]. Available: http://forge.gridforum.org/projects/drmaa-wg/

[12] H. Rajic, R. Brobst, W. Chan, F. Ferstl, J. Gardiner, A. Haas, B. Nitzberg, H. Rajic, and J. Tollefsrud, "Distributed Resource Management Application API Specification 1.0," Global Grid Forum, Tech. Rep., June 2004, GFD.022. [Online]. Available: http://www.ggf.org/documents/GFD.22.pdf

[13] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee, and H. Casanova, "An Overview of GridRPC: A Remote Procedure Call API for Grid Computing," in *3rd International Workshop on Grid Computing*, vol. 2536. Springer-Verlag, Lecture Notes in Computer Science, November 2002, pp. 274–278.

[14] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, C. Fabianek, T. Hiroyasu, E. Meek, M. Miller, K. Sagi, K. Seymour, Z. Shi, and S. Vadhiyar, "Users' Guide to NetSolve V2.0," The NetSolve Project, Innovative Computing Laboratory, Department of Computer Science, University of Tennessee, Tech. Rep., 2004. [Online]. Available: http://icl.cs.utk.edu/netsolve/

[15] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka, "Ninf-G: A reference implementation of RPC-based programming middleware for Grid computing," *Journal of Grid Computing*, vol. 1, no. 1, pp. 41–51, ???? 2003. [Online]. Available: http://ipsapp009.kluweronline.com/content/getfile/6160/1/3/abstract.htm;http://ipsapp009.kluweronline.com/content/getfile/6160/1/3/fulltext.pdf

[16] E. Caron and F. Desprez, "DIET: A Scalable Toolbox to Build Network Enabled Servers on the Grid," *International Journal of High Performance Computing Applications*, 2005, to appear.

[17] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee, and H. Casanova, "A GridRPC Model and API for End-User Applications," September 2005, http://www.ggf.org/documents/GFD.52.pdf.

[18] GGF. (2003) Grid Checkpoint Recovery Working Group. [Online]. Available: http://forge.gridforum.org/projects/gridcpr-wg/

[19] GGF. (2004) Grid File Systems Working Group. [Online]. Available: http://forge.gridforum.org/projects/gfs-wg/

[20] Resource Namespace Service - Proposed Final Draft v1.3 . [Online]. Available: http://forge.gridforum.org/projects/gfs-wg/document/Resource_Namespace_Service_-_Proposed_Final_Draft_v1.3/en/2

[21] GGF. (2004) ByteIO Working Group. [Online]. Available: http://forge.gridforum.org/projects/byteio-wg/

[22] ——. (2001) GridFTP Working Group. [Online]. Available: http://forge.gridforum.org/projects/gridftp-wg/

[23] GGF. (2004) Job Submission and Description Language Working Group. [Online]. Available: http://forge.gridforum.org/projects/jsdl-wg/

[24] GGF. (2003) Persistent Archives Research Group. [Online]. Available: http://forge.gridforum.org/projects/pa-rg/

[25] ——. (2004) OGSA Replication Services Working Group. [Online]. Available: http://forge.gridforum.org/projects/jsdl-wg/

[26] R. Moore and A. Merzky, "Persistent Archive Concepts," Global Grid Forum, Tech. Rep., December 2003, GFD.026. [Online]. Available: http://www.ggf.org/documents/GFD.26.pdf

[27] GridLab Project. (2002) The Replica Management Service. [Online]. Available: http://www.gridlab.org/data

[28] I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *International Journal of Supercomputer Applications*, vol. 15, no. 3, 2001.

[29] K. Davis, T. Goodale, and A. Merzky, "GAT API Specification: Object Based," *EU Project GridLab, Deliverable D1.5*, June 2003. [Online]. Available: http://www.gridlab.org/WorkPackages/wp-1/Documents/Gridlab-1-GAS-0004.ObjectBasedAPISpecification.pdf

[30] G. von Laszewski, I. Foster, J. Gawor, and P. Lane, "A Java Commodity Grid Kit," *Concurrency and Computation: Practice and Experience*, vol. 13, no. 8–9, pp. 643–662, 2001, http://www.cogkits.org.

[31] D. C. D. Lawrence Berkeley National Laboratory, Computational Research Division. (2005) Python Globus (pyGlobus). [Online]. Available: http://dsd.lbl.gov/gtg/projects/pyGlobus/

[32] S. Jha, P. V. Coveney, M. Harvey, and R. Pinning, "SPICE: Simulated Pore Interactive Computing Environment," *Proceedings of ACM/IEEE Supercomputing Conference 2005*, 2005, http://sc05.supercomputing.org/schedule/pdf/anal109.pdf.

[33] "The TeraGyroid Experiment", Accepted for GGF10 Applications Workshop. [Online]. Available: http://www.realitygrid.org/TeraGyroid-Case-Study-GGF10.pdf,http://www.realitygrid.org/TeraGyroid-Case-Study-GGF10.pdf

[34] R. Berlich, M. Kunze, and K. Schwarz, "Grid computing in Europe: from research to deployment," in *CRPIT '44: Proceedings of the 2005 Australasian workshop on Grid computing and e-research*. Darlinghurst, Australia: Australian Computer Society, Inc., 2005, pp. 21–27.

[35] S. D. Team. (2005) Simple API for Grid Applications – Strawman API Version 0.2. [Online]. Available: http://wiki.cct.lsu.edu/saga/space/start/strawman-api-v0.2.pdf

[36] S. Pickles, R. Pinning, A. Porter, G. Riley, R. pert Ford, K. Mayes, D. Snelling, J. Stanton, S. Kenny, and S. Jha, "The realitygrid computational steering api version 1.1," RealityGrid, Tech. Rep., 2004, e-print.

[37] P. Kacsuk, G. Dózsa, J. Kovács, R. Lovas, N. Podhorszki, Z. Balaton, and G. Gombás, "P-GRADE: A Grid Programming Environment," *Journal of Grid Computing*, vol. 2, pp. 171–197, 2003.