GWD-I.71
SAGA-RG

Shantenu Jha, University College London
Andre Merzky, Vrije Universiteit, Amsterdam

May 09 2006

A Requirements Analysis for a Simple API for Grid Applications

Status of This Document

This document provides information to the GGF Applications Area in the Grid Standards council. It does not define any standards or technical recommendations. Distribution is unlimited.

Abstract

This document distills the use cases [6] received by the Simple API for Grid Applications research group (SAGA-RG) and extracts the salient features into a set of requirements for the API In addition to the requirements drawn from the use cases, by analysing related ongoing developments in the grid community, this document tries to define further the scope and requirements of any simple API for applications.

Contents

1 Introduction

The GGF's SAGA Research Group[1] strives to define a high level API that addresses directly the need of application developers. This document outlines the main features that such an API should capture and in the process guides the scope, granularity and design of such an API. The bulk of these features are derived from an analysis of the use cases that the SAGA-RG receieved [6]; the remaining features discussed are derived from an analysis of the status and trends in related areas of grid computing.

A useful though informal way of understanding SAGA's aim and scope is to think of the MPI standard. We believe that SAGA will do for grid applications what MPI did for parallel applications: it will extract the correct semantics and right level of abstraction, so as to permit the writing of portable applications (across different middlewares) using a simple, minimally complete, consistent, uniform set of function calls and thus be widely usable. All this, while at the same time enabling the application to be coded in a manner that keeps it independent of the underlying infrastructural details.

It is also very useful to mention explicitly that SAGA API is for coding *against* a middleware and not for coding middleware directly. Admittedly, the term *application* is an overloaded term and means different things to different people, however for the purposes of this document (and understanding SAGA in general) it is useful to remember that the applications in scope and of interest to SAGA

[1] The group is currently undergoing a transformation: the SAGA RG is changing focus to be an umbrella group, tasked primarily with spawning smaller, focussed working groups which in turn will be ultimately responsible for the API. The umbrella research group will ensure a consistent look and feel across the WG and in will addition coordinate SAGA's activities with other groups within the GGF.

as well as being potential users of SAGA are those most typically used by the *end-user*.

This document distills the use cases [6] received by the Simple API for Grid Applications research group (SAGA-RG) and extracts the salient features into a set of requirements for the API[2]. In addition to the requirements drawn from the use cases, by analysing related ongoing developments in the grid community, this document tries to define further the scope and requirements of any simple API for applications.

This document is divided into three main sections. In Section 2 the salient points and primary conclusions from the submitted use cases are presented. We then change perspectives and utilise the use cases to motivate a discussion of functional and non-functional requirement areas of relevance. The aim of this section is to capture and prioritize the functional and non-functional areas which in turn guide the design and implementation of the SAGA API. Section 3 contains an analysis and discusses the main lessons learnt from projects that implemented frameworks that are based upon the SAGA philosphy of grid enabling applications via user-level API. This section also discusses gLite which is an instance of a distinct approach to programming applications for grid environments. A discussion of SAGA's relation vis-a-vis other GGF groups and middleware development projects is presented in Section 4. This section closes with remarks on portability and language binding issues relevant to SAGA.

As alluded to, the development of the API is guided by the use cases received; however it is important to realize that there is still a need for more use cases, especially from non-academic projects and application developers. These use cases in turn may possibily influence the design of a version 2.0 of SAGA.

1.1 Target Audience

The document targets primarily the SAGA Research Group, and is supposed to both document and guide the work performed in the group. In particular, the scope of the SAGA API is primarily derived from the findings in this document. The target audience however, is not confined to the SAGA Research Group. It is also instructive for designers, developers and implementors of independent efforts such as gLite [2], Superscalar [10, 11] and GAT [1].

2 Requirements from SAGA Use Cases

The SAGA Research Group elicited a number of use cases to guide its API specification process. These use cases are published as GGF document *"GGF*

[2]These requirements are supposed to drive the API development, however given that the group has had a design team put together a strawman of the API, which has been well received, this document will instead allow a sanity check of the strawman.

SAGA-RG Use Cases" [6]. This section discusses the use case template, main features of the use cases, and then analyses the implications for the SAGA API.

Deriving Requirements: Some General Remarks

It was a rather simple exercise to derive the set of requirements for most of the considered use cases. This speaks, we think, for both the design of the use case templates and of the work the use case authors took to provide the use cases. The next subsection 2.1 enumerates the functional and non-functional areas and annotates them – which helps the interpretation of the less obvious requirements.

Subsections 2.2 and 2.3 list the requirements – both functional and non-functional – derived from an analysis of the use cases and discusses them, from which a set of recommendations for the SAGA API design are derived.

A histogram of the functional and non-functional areas from the use cases is presented in Table. 2.2. The frequency count needs careful interpretation though. For example, the frequency is not weighted by *importance*; sometimes an area is very central to a use case, sometimes it is barely touched, but both are treated the same for the histogram. Also not reflected in the frequency count is the fact that an area is sometimes not explicitly mentioned, but might well be useful to the use case (e.g., 'events' for visualization use cases). There is also the issue that some areas have some level of functional overlaps (i.e., steering and events).

The SAGA-RG received nine use cases from the GridRPC Working Group; for the sake of this analysis, these have been pooled together into a single use case, due to their similarity. Hence, the respective areas are counted only once. Currently, a dedicated design team in working in the SAGA-RG to derive more detailed requirements for an integration of RPC into SAGA, and to draft a RPC focused strawman API. are counted only once.

Functional Requirement versus Non-functional Requirements:

The separation into functional and non-functional areas is based more on a classification scheme than a rigid definition. Thus, as with most classification schemes, it is not water tight and there is scope for some ambiguity about which category a particular requirement might fall into. Functional requirements are loosely speaking those for which a direct API call (or calls) exists, e.g., job management. Non-functional requirements do not have explicit call(s) associated with them, though they may influence the syntax and semantics as well as the design and implementation of specific API calls and maybe even the API as a whole. For example, the requirement for security and bounds on Quality-of-service (QoS), do not require explicit calls, but do need to be supported by the API. Auditing is an example of a possible requirement that could fall into both categories.

Requirement Levels and RFC 2119:

The use of the requirement levels from RFC 2119 [3], such as "must", "should",

"may", and their negations, has been investigated in the course of writing this document. The authors, however decided not to use these terms as they were found to be not appropriate for expressing different levels of requirements *on* a technical (here: API) specification. The key words from RFC 2119 have been defined for use *within* a specification, and they only make sense in such a context. In the following, only the word "should" is used to express whether or not a certain functional or non-functional property should be met by a SAGA specification.

2.1 Use Cases in Detail

The SAGA Use Case Template

The use case template was aimed at an audience that consisted of projects as well as specific applications, that had either developed APIs/interfaces to facilitate grid applications, or those that felt that a SAGA like API would make grid-enabling applications easier, robust and extensible. The use case template was drafted so as to facilitate the extraction of information that would inform the initial scope and design of a SAGA API. After initially gathering general information about the projects and applications of interest, the template sought information on actual usage scenarios of the applications. Information on the typical users and resources required by the applications was also requested. Resources in this context ranged from software and hardware constraints to services required for running the application in a distributed environment.

The issues of security, performance, typical resource selection and runtime environments were queried. The audience was then asked to enumerate all grid technologies used and importantly were asked how often the applications in scope were used in a grid-context. Finally the respondents were asked to provide pseudo-code for some prototypical API calls that would illustrate their requirements.

The usecases are archived and online accessable at the GGF Use Case Repository [4], and also published as an information GGF document [6].

CoreGrid Use Case

The use case has a broad scope, and explicitly lists both functional and non functional areas which are expected to be covered by the SAGA API. Resource discovery and selection are listed, but interpreted as being required at the middleware level and not at the application level. Integration with commodity techniques, however is a noteworthy additional requirement. Also noteworthy is the requirement for synchronous event notification.

Coastal Modeling Use Case

Relative to CoreGrid this presents a rather specific usage scenario. It describes an ongoing project and lists a wide range of requirements. It is not easy to prioritize the relative importance of the many requirements. "Must have" requirements appear to be those of data access, management, and remote visualization. There is a stated requirement for QoS – although it is somewhat unclear if they need to be exposed at the API level, e.g., the API might possibly provide a means for specification, but not necessarily negotiation of QoS.

DRMAA Use Case

Although the two have very different scopes, the DRMAA Working Group [5, 9] and SAGA Research Group share a common aim – that of providing a high level API. That motivates the SAGA API to adhere to the API structure defined in DRMAA for job submission. It further motivates the consideration of bulk operations as a major performance obstacle in distributed systems. It describes bulk job submission, but the argumentation of the use case authors seems applicable to other bulk operations than just job submission, as well.

DiVA Use Case

This very elaborate use case describes the design of a grid distributed component system, which can be assembled into a visualization pipeline. Security, QoS, resource and service discovery, and data streaming are the cornerstones of the DiVA requirements. There is an implicit requirement for asynchronous operations and notifications (although both could be provided outside SAGA). It is interesting to note, that if the performance requirements of the use case were fully respected in SAGA, it would imply a very low latency throughput overhead of the API implementation (zero copy), at least for streaming and event notification.

GridLab Use Case

Notable for this use case is the paradigm of self-awareness for a Grid application: the application can handle itself as a job instance, clone itself, spawn children etc. (`saga::self::get-job-description`). The use case further motivates the use of application level monitoring and asynchronous notification.

KoDaVis Use Case

This use case focusses primarily on data management and visualization. However, interesting is the expressed need for an application level information ser-

vice. Interstingly, this appears to be the only use case (UC) that has asked *explicitly* for information services.

Medical Imaging Use Case

The GEMSS use case is unique amongst received UC as it focuses on resource reservation, management, advanced reservation and QoS negotiations. These are all advanced requirements, and traditionally not often exposed traditionally at the API level.

RealityGrid Use Case

The RealityGrid use case adds important non-functional requirements, such as API stability, scalability, and the need to support a wide range of Grid middleware. The single main functional requirement is computational steering. Computational steering however, requires several other functions, e.g., migrations, checkpointing and monitoring. Additional functional areas are job submission and data management.

Superscalar Use Case

Superscalar is special in the respect that it describes a middleware or tool set, not a scientific application. However, its requirements match the level and scope of the other use cases very well, and focuses on job and data management. Asynchronous and synchronous notification are prominent non-functional requirements.

Visit Use Case

This use case focuses on application level steering and communication, whereby large amounts of data are to be streamed between components. Asynchronous operations and notification seem to add a lot of flexibility to that use case.

Visualization Service Use Case

The visualization service described in this use case opens a completely new aspect for the SAGA API, as it describes the access to a custom remote service. That functionality is comparable to the automatic creation of client side stubs for a WSDL description. In some respect that area is covered by GridRPC like methods (see below). However, interesting for SAGA are service and resource discovery aspects, as well as steering and asynchronous notification.

LSU Viz Service Use Case

The use case describes a block oriented message API with asynchronous notification, and motivates its use for remote visualization of large data sets.

GridRPC (set of) Use Cases

The GridRPC working group in GGF applied the SAGA use case template to a number (9) of their own use cases, and submitted those to the SAGA group. In fact, these use cases match the SAGA problem space very well. As they are very similar in terms of scope, they are, for the sake of this document, treated together.

The use cases motivate (not surprisingly) the utilization of Remote Procedure Calls for Grid applications. That goes along with a set of requirements for asynchronous operations, resource discovery and data management.

2.2 Functional Areas covered by the Use Cases

The functional areas identified in the submitted use cases are:

1. **Job Management:** Submission and management of jobs. Individual resources are not necessarily specified, or only identified by name and job requirements.
2. **Resource Management:** Allows for fine grained description and selection (discovery) of resources to be used for job management.
3. **Data Management:** Management of files as entities (copy, move, ...), does not include access to file contents, nor replica management.
4. **Data Access:** Access to contents of files
5. **Logical Files:** File replica management
6. **Streams:** Communication between running processes, with mechanisms similar to BSD streams
7. **Data Bases:** Access to remote data bases, no particular schema implied
8. **Events:** Short message style events as used for inter process communication, synchronization etc.
9. **Steering:** Support for steering of parameters of remote applications
10. **Information Services:** Read and write capabilities to persistent information repositories, with support for application specific information storage

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

11. **Communication:** Any means of communication not covered above, as large data messages and RPC

2.2.1 Discussion of Functional Areas

Job Management: The scenarios from the use cases cover most importantly job submission, and tracking of job status. Additionally, most use cases classify this requirement as a "must have".

Jobs in the use cases are often described by RSL or JSDL like languages. The most commonly used attributes in addition to executable name and parameters, are environment variables and input/output files (which sometimes require staging).

A number of use cases expect the API to allow various actions performed on the jobs, which mostly change their state, such as: `suspend()`, `continue()`, `kill()`, `signal()`, and `migrate()`.

- *Job Submission and Management should be included in the very first SAGA API.*

Resource Management: A number of use cases need the specific capability of being able to select resources. In addition some projects and applications also need the ability to discover suitable resources, before selecting and submitting to resources.

It is important to note that there are ongoing developments in resource management [8], that could make it more meaningful to keep resource management out of the API for some applications (i.e. away from the application level). It is illustrative to note for example that GridRPC and MPI do not provide an

interface for resource management and so one could in principle argue neither should SAGA.

But based upon use cases received, it is currently felt that the ability to manage resources is required (and as a "must have") at the API level in these use cases.

- *Resource discovery should be supported by the SAGA API.*

Data Management: Navigating remote file directories structures and manipulating the location of files in these structures are intrinsic parts of many use cases. Simple operations such as `list()`, `mkdir()`, `copy()`, `move()`, and `remove()` are required; sometimes `find()` and iterators for large directories seem useful.

- *Data Management should be supported by the SAGA API.*

Data Access: In addition to the requirement of data management, access to the *content* of individual remote files is often required. The trinity *read /write/seek* fulfills most of these use cases, however, some performance considerations seem to imply other file access paradigms (although they are not explicitly mentioned in the use cases received). SAGA should not however, preclude efficient data access.

- *Efficient Data Access should be supported by the SAGA API.*

Logical Files: Only few use cases requested the support of replica systems. This was surprising, as replica systems seems to be well accepted in the grid community, and are amongst the more stable and widely deployed elements of todays grids. This might either reflect a bias in the use cases received, or might be a consequence of the fact that the use cases authors are currently not working with replica systems, but could possibly utilize them anyway, if the API provided appropriate support. The set of capabilities however, expected from replica systems is small, e.g., management of the set of replicas for each logical file, including replication (creation of new replicas). The support for meta data (and search on them) for logical files is also mentioned explicitly in some of the use cases received.

As data replication features are amongst the most accepted grid paradigms, it is recommended that at the very least, basic support for it be included, even if it is not required by the majority of use cases.

- *Data Replication should be supported by the SAGA API.*

Information Services: A surprisingly large number of use cases asked for the support of application level information repositories. Those included some use cases which wanted to attach meta data to logical files. Others wanted to exchange (and persistently store) application specific meta data. The meta data in question seem to be mostly lists of simple key/value pairs – however, the set of keys and the set of value types is not predetermined, and application dependent.

- *Persistent storage of application specific information should be supported by the SAGA API.*

Streams: As a means of exchanging larger amounts of data between applications, BSD like streams seem to be the favorite paradigm listed in the use cases. In particular the various remote visualization scenarios require support for remote data streaming, which allows for simple end point authorization, and handles fire walls and other grid specific problems transparently. Connection setup (client/server bootstrapping) and `read()`/`write()` seems to serve most use cases, however, asynchronous notification on incoming data (select) seem crucial for several applications.

- *Streaming of data should be supported by the SAGA API.*
- *Asynchronous notification should be supported by the SAGA API.*

At a more abstract level, most of the streaming use cases seem to exchange *messages* at the application level, i.e., larger independent chunks of data with intrinsic structure. One use case specifically requested support for messages at the API level, and in fact, that paradigm would seemingly support (and simplify) the other streaming use cases as well.

- *Support for messages on top of the streaming API should be considered by the SAGA API.*

Events and Steering: Along the same lines, various use cases benefit from asynchronous and timely delivery of custom events – e.g., for synchronization of multiple processes. Several use cases explicitly list steering as the usage scenario for such events.

- *Asynchronous notification should be supported by the SAGA API.*
- *Application level event generation and delivery should be supported by the SAGA API.*
- *Application steering should be supported by the SAGA API, but more use cases would be useful.*

Communication: The communication schemes above (streams, events, steering) support not all use cases with need of remote communication. Some of the remote visualization use cases require a more high level communication scheme than streams for data exchange (send/receive of large message buffers, see above). Also, a significant number of use cases are very specifically requesting support for GridRPC (these use cases have been submitted by the GridRPC group in GGF). In this analysis, those requirements are subsumed in the 'Communication' area. However, apart from GridRPC it is currently not possible to pinpoint more specific communication schemes which need supporting – more use cases are required for this.

- *GridRPC should be supported by the SAGA API.*
- *Further communication schemes should be considered as additional use cases are submitted to the group.*

Data Bases: Only one use case requested API support for data base access, and that included a very specific data base layout. We think that before SAGA addresses data base access, more and more specific use cases are required.

- *Data Base access does currently not require explicit support in the SAGA API.*

2.3 Non-Functional Areas covered by the Use Cases

The identified non-functional areas are listed below. They do not necessarily need a representation in the API in the form of additional method calls – however, the API specification needs to be aware of these areas, and should allow application to exploit these functionalities. Some areas, such as transactions, may have no reflection in the API at all; others, such as tasks, may influence the overall look and feel of the API.

1. **Bulk Operations:** Support large numbers of very similar or identical remote operations efficiently
2. **Security:** Allow or require support for secure infrastructure (e.g., support credential management)
3. **Error:** Have fine grained and verbose support for error handling, e.g., for the sake of error recovery and debugging
4. **Quality of Service:** Support the notion of QoS on various levels, e.g., for deadline scheduling, bandwidth reservation etc.
5. **Auditing:** Allow for audit traces of all remote operations

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

6. **Transactions:** Support remote operations as transactions
7. **Workflow:** Include support for work flow on API level (e.g. allow to specify job dependencies)
8. **Asynchronous Operations:** Allow for asynchronous operations

2.3.1 Discussion of Non-Functional Areas

Asynchronous Operations: Although these areas are not amongst the most requested, we consider them to be of importance. Asynchronous operations are very crucial to remote operations, as those have usually no guaranteed and potentially very long and varying response times – very slow responses are difficult to distinguish from failures and timeouts. Grid applications definitely require support for asynchronous calls.

- *Asynchronous Operations should be supported by the API.*

It is prudent however, to not make every call asynchronous, but to have a clear separation between asynchronous and synchronous calls. Having asynchronous versions and synchronous versions which helps keep the complexity of the commonly used calls low, appears to be the right approach.

Bulk Operations: For bulk operations we received a very specific use case. Also, many common distributed computing techniques (e.g. parameter sweeps) can certainly benefit from bulk operations. Also, for many individual remote operations, latencies and response time can add up to unreasonable long overall response times – bulk operations (i.e., clustering of multiple remote requests into a single one) are one way to avoid that. In the distributed community, bulk operations and asynchronous method calls are well understood.

- *Bulk Operations should be supported by the API.*

Errors: support for good error reporting seems to be self-evident for every API design. However, it seems particular important to distinguish several error types. For illustration:

```
file.copy ("http://letalhost//tmp/file.dat", ".");
```

The call above might fail for a plethora of reasons. However, it should be possible to recover from some of them (e.g. service not available, timeout) by retrying later, but it seems unlikely that it is possible to recover from the mis-spelling (localhost → letalhost). Whether an error is recoverable or not, depends on many things: the specific implementation of the API, the middleware it binds to, the policy of the application etc.

- *The error support of the API should allow for **application level** error recovery strategies.*

Security: Most use cases have at least some security requirements – however, they are often not very specific, and mostly say *"should be secure"*. We think that this reflects two points: firstly, users are willing to use whatever security infrastructure is available and/or required; and secondly, users do not have much experience with security, and are probably not willing to learn too much about it, unless absolutely necessary.

- *The SAGA API should be implementable on a variety of security infrastructures.*

- *The SAGA API should expose only a minimum of security details, if any at all.*

SAGA might possibly target ACLs for files and name spaces. Unfortunately however, at the time of writing (and development of version 1.0 of the SAGA specification) it is still unclear how best to introduce and handle the issue of security. Despite ongoing discussions with the security area in GGF, no clear picture of either end user demands nor common middleware paradigms for security emerged. We think this is a testament to the innate difficulty of the security problem.

Auditing: Logging, bookkeeping, auditing and accounting are frequently cited as crucial for production level deployment of Grids, and are also listed in several use cases. However, most use cases seem to require the *existence* of these features at the implementation or middleware level, but do not seem not to need *access* to them at the API level. More specific use cases would be required to provide access to these features on API level.

- *Auditing, logging and accounting should not be exposed in the API.*

Workflow: Grids seem to provide optimal environments for complex workflows, and many projects and research groups are working both on workflow specifications and execution environments. Workflow is also listed in a (small) number of SAGA use cases – however, a need for support of workflows at the API level seems not required by any of them right now. Instead, support for workflow seems to be required on middleware level.

- *Workflows do not require explicit support on API level.*

Quality of Service: Several use cases specified a need for Quality-of-service negotiations. The use cases present specific and varied constraints, e.g. the need for specific time lines for remote executions to be kept, have specific bandwidth requirements, requirements to remote operations reliability etc. However, the diversity of QoS problems covered makes it difficult to come up with a conclusion on *how* QoS should influence the API – e.g. many additional parameters could be thought of for the file copy method to specify its QoS needs, but the same could be said for all other calls, and the API would be cluttered in no time.

Given that there are at least three (strong) use cases however, that request some form of Qos, the topic should definitely addressed at some point. Unless more specific QoS use cases are available to SAGA however, QoS should be considered as a future, generic extension to the API, e.g. as a set of QoS related attributes to objects or tasks. Maybe a "QoS context" which would in turn prevent the API from being cluttered is an option worth considering.

- *QoS does not require explicit support on API level. This issue should be revisited for SAGA version 2.0.*

Transactions: Surprisingly, not a single submitted use case placed an explicit requirement of transactions for remote operations. Once again, this might just be a reflection of the limited number (and scope) of use cases received.

- *Transactions do not require explicit support at the API level.*

3 Requirements from Other Implementations

A number of current and past projects in the area of Grid middleware and Grid applications cover or touch the same area as the SAGA group. There exist a number of APIs and interfaces, usually targeting a subset of the SAGA audience.

This section reviews these developments, and, from their experiences, derives a set of non-functional requirements for the SAGA API specification.

3.1 GAT

The *Grid Application Toolkit (GAT)* is an application level Grid API, and targets a user group similar to SAGA. This section describes lessons learnt in the design process and from the implementation of the GAT. We derive several points we feel should be used as requirements for SAGA, i.e., the 'lessons learnt' in GAT. Given the similarities in the scope, the lessons learnt from GAT are also applicable to SAGA – and thus we highlight several points that serve as requirement for SAGA too.

GAT Design

The GAT API design followed an object oriented (OO) approach. It was felt that a mapping of an OO API to procedural languages would be easier than mapping a procedural API to OO languages. This proved a valid approach, as the later implemented language bindings in both procedural (C) and OO languages (C++, Java and Python), which are equally well accepted.

- *The SAGA API Specification should be Object Oriented.*

Also, the usage of asynchronous notification mapped very well to various language bindings of GAT, and gave no unreasonable trouble to implementers. On the other hand, they increased the simplicity and usability of the API significantly (in fact, a major demand on the GAT today is to extend its asynchronous capabilities).

- *Asynchronous notification should be supported by the SAGA API.*

GAT API Scope

The scope of GAT API was derived from a very limited set of use cases internal to the project. The final GAT API served these use cases very well, but to some extent failed to enable additional use cases.

- *The range of motivating use cases should be as wide as possible, and as narrow as necessary to agree on a finite scope.*

The most heavily used parts of the API span *File Management*, *Replica Management*, *Resource discovery* and *Job Submission*, and the *Advert Service*. The latter provides an interface to persistent storage for arbitrary information, and also for serialized GAT objects.

- *Application level persistent information exchange increases the convenience of implementing Grid application scenarios significantly.*

Persistent information exchange was not derived from a specific use case, but created in order to allow middleware independent, persistent exchange of custom information. It is well accepted by GAT users, and heavily used. However, the GAT team learned that the learning curve for new paradigms must not be steep – otherwise adoption is severely limited.

- *Frequently used paradigms should be the most powerful ones.*
- *New paradigms should be the most simple ones.*

GAT Implementation

The GAT is implemented in C and in Java, and provides wrappers around the C implementation for bot C++ and Python. Adaptors (i.e., middleware bindings) can be written in in C, C++ and Python (for the C implementation) or Java (for the Java implementation). In particular the ability to provide middleware bindings in various languages proved very useful. In respect to the API *specification*, however, we did not encounter any facet which impacted the implementability of the API significantly.

- *Multiple language bindings for the API are essential.*
- *The ability to support diverse middleware technologies and programming back ends is essential for the implementation, but of no concern to the API design.*

The single most asked feature of the various GAT implementations has been *Documentation*, both as references, and also as tutorials and examples.

- *The success of the API will stand and fall with the quality of its specification and documentation.*

3.2 CoG / pyGlobus

CoG Design

Although Globus has a powerful API, it is not an API that, due to its complexity, is of direct utility for an application developer. Thus there is a need of a wrapper layer that exposes the features of Globus (or for that matter any grid middleware) at a level of abstraction natural to application developers. The *Commodity Grid (CoG)* is such an application oriented API, which wraps the various incarnations and versions of the Globus API. The CoG effort started

very early, within the Globus project. The design of the CoG changed over time, from a thin wrapper, to a re-implementation of parts of Globus, to a more generic, flexible environment on top of Globus and other middleware. The CoG comes in two flavors, Java (Java-CoG) and Python (pyGlobus). The Python CoG, which is external to the Globus project developed at LBNL, sticks to the 'thin-wrapper' design of the earlier Java-CoG.

CoG API Scope

The CoG, by design, strives to provide the same set of capabilities covered by the Globus middleware. That includes job management on remote resources, data management via GridFTP and the Globus Replica Service, access to the Grid Information Service etc. The use cases covered by CoG are similar to those targeted by the Globus project in general – however, the CoG intends to make their implementation much simpler (rapid prototyping is a declared objective).

The stability of the CoG API (relative to the stability of the Globus APIs) and the isolation from the Globus release cycles are perceived as major advantages for using the CoG instead of coding against Globus directly.

- *APIs should isolate developers from versioning and diversity of underlying layers.*

The latest CoG version comes with support for tasks, task dependencies, work flow and other abstractions, which have no one-to-one equivalent in the underlying Grid Middleware. However, as there seems to be a clear demand for these abstractions from the user community, they are well accepted.

- *Additional abstractions and paradigms provided by the Java-CoG are, where they serve well motivated use cases, welcomed by application developers and constitute added value.*

CoG Implementation

The previous observations focused mainly on Java-CoG. Both variants however found a very broad user base (also outside the traditional Globus use community), as both languages are very useful for rapid prototyping. Also, they are well suited for Grid environments, as they do not need compilation, which makes application deployment much simpler.

- *The availability of the CoG for Languages other than C/C++ is a very successful feature.*

3.3 RealityGrid

RealityGrid is primarily concerned with typically large scale applications that aim to use a grid infrastructure for computational steering. Computational steering is a term commonly used to cover a wide range of features, for example the ability to change a simple parameter or the ability to spawn a simulation when there is a scientific reason to. Computational steering is a compound functionality, in the sense that it requires several simpler functionality to be available. For example, computational steering might make use of features like launch jobs, file staging or more complex features like checkpointing, monitoring, dynamic resource allocation and QoS constraints. Consequently, computational steering appears to be a functional area for which a standard interface should be developed only after the areas that it depends on have been developed and stabilized.

There are several distinct applications that require computational steering. Various applications are written in different languages, but more importantly use different programming paradigms (MPI, PVM etc.). Consequently, a common interface that can utilize the grid infrastructure but be used across different programming paradigms is critical. Typically application codes are the result of many years of development and there often is not sufficient expertise to support active re-engineering or development; consequently as few changes and as infrequently as possible is the desired. Consequently, once an application has been interfaced to a steering framework, it is highly desirable if it can be reused. The above provides justification for a common, consistent API. The following is strong motivation to add "simple" to the list of API attributes. Being able to quickly engineer a specific code so as to implement functionality to be computationally steered is important to the application scientists so as to keep the barrier low.

In addition, there is a need for sufficiently generic and versatile API which arises from the fact that multiple codes from several application domains should ideally be able to use the same API.

Once an application code has been extended to provide it a computational steering feature, the same code should be usable in all computing environments – grid, parallel or single processor environments. Refactoring for different or specific environments is highly undesirable and thus an API that provides computational steering (or in general any feature) should be usable across all possibly desirable computing environments. But this may come at an additional compile/run-time complexity. (But possible work arounds like plugins may help provide a compromise between single development tree and easy of deployment).

- *Quick and easy and generic APIs need to be usable across all commonly used scientific computing environments.*

3.4 gLite (EGEE)

The gLite web site states:

> *"gLite [...] is the next generation middleware for grid computing. [...] gLite provides a bleeding-e.g. best-of-breed framework for building grid applications tapping into the power of distributed computing and storage resources across the Internet."*

gLite represents a Grid middleware, comprised of a number of services tailored to the specific, but broad needs of the EGEE eScience project community. Interesting for this document is that gLite comes with an Application Interface (a WSDL), which allows easy access to the complete gLite middleware from within applications. The WSDL is, by design and intent, similar to what SAGA tries to accomplish. However, that WSDL defines a web service interface, not an API. That web service dispatches the Application requests to the particular gLite middleware services.

That interface allows the creation and management of security certificates, supports access to data and replica management services, and job submission and (workload) management services. It also allows for POSIX-like remote file I/O. gLite further provides accounting, logging, and bookkeeping. Service discovery is also exposed at the application level.

Defining an API is not the only way to serve application developers – tools and high level services can complement the SAGA API efforts. That should be kept in mind while defining the SAGA scope: some problems might best be approached with a different tool set than APIs, keeping the SAGA API focused and *S*imple. gLite's 'Grid Access Service' is a notable example. 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.

- *The SAGA API should be careful to not try to address problems that might be better solved using alternative approaches. Also, it is important to determine which problems are not effectively solved using an API approach but competing approaches.*

4 Infrastructure Requirements Assumptions

4.1 Relation to Other GGF Groups

The SAGA API, by design, touches a number of areas which are already covered by efforts of other groups in GGF, such as for example the API for job submission is covered by the DRMAA Working Group.

As the scope of the SAGA API is initially very limited, the number of groups affected by the SAGA API specification is limited as well. These groups are listed here, and actions for synchronization with these groups are recommended[3]. It should be noted that interactions with these groups have been ongoing since the formation of the SAGA group, and that the SAGA group members are very happy and satisfied about the involvement and support from the respective GGF groups. It is worth mentioning that a more formal document *"SAGA use of other Grid specifications"* will be produced by the SAGA-RG. This document will survey and evaluate SAGA reference implementations with underlying models of grid-middleware including, but not confined to OGSA. We hope this will help elaborate further SAGA's relationship to other GGF groups.

Security

As described earlier, SAGA has received use cases with explicit security requirements. However, the work of the SAGA-RG is currently security model agnostic due to ignorance. We seek input from GGF Security Area, as the right level of abstraction for Grid security models is currently unclear to SAGA, and to other groups working on Grid API in GGF (see also notes about security in section 2.3).

- *Synchronization with the GGF security area about API related security paradigms is essential, and should be continued.*

Resource management and Job Submission

Management and utilization of resources is of course a central topic in Grids, and a large number of groups in GGF touch that topic one way or the other. Most notably, the DRMAA-WG defines a high level API for job submission, the JSDL-WG defines a XML schema for job definition, and the OGSA BES-WG defines a basic execution service for remote job submission.

- *The SAGA-WG should synchronize the paradigms exposed by the SAGA API with those defined by the high level work of the DRMAA, JSDL and OGSA BES working groups.*

The OGSA and WSRF related middleware groups in GGF deals with the specifics of resource utilization on a lower level, and are of no immediate concern to the SAGA API specification. However, as the paradigms developed and exposed by these groups are very central to the future Grid technologies, a certain amount of synchronization, and check for paradigm compatibility (i.e., implementability) on OGSA/WSRF based middleware seems necessary.

[3]Current negotiations about an extended SAGA scope in a re-chartered SAGA umbrella RG will require a thorough review of this list.

- *Although the SAGA API specification seems unrelated to the low level efforts of OGSA/WSRF, the paradigms defined and provided by these groups should be considered and frequently checked for compatibility.*

A number of WGs at the GGF are tackling the problems of resource discovery and, of information services, e.g. CIM. That problem space is, for now, outside the scope of the SAGA Strawman API (see discussion in the summary). It is possible that at some stage, if SAGA API has a responsibility to address a wider scope, there will be a need to review SAGAs relationshop to groups such as CIM. Naturally, any scope extension of the SAGA API should imply the review of the SAGA-WG relations to groups such as CIM.

Name Spaces

The SAGA API will touch the topic of name spaces on various levels, e.g. for physical and for logical files. In particular the Grid File System WG in GGF covers that problem field in respect to paradigms, definitions and service architecture.

- *The SAGA-WG should synchronize their notion of name spaces with those defined by the GFS-WG.*

Files

The area of management of and access to is covered by several groups: GridFTP, ByteIO, and DAIS[4]. The GridFTP group, working on wire level, is probably of no direct concern to the API specification, however, there are lessons to learn about performance issues of grid file management and data access.

- *GridFTP should not influence the SAGA API specification directly, however, their findings about performance issues for remote data management and access should be taken into consideration.*

The ByteIO-WG defines a WSRF interface for binary access to remote files. As such, the groups goals are very similar to the SAGA goals, although their specification aims at a different technology level.

- *The SAGA-WG should synchronize their data access paradigms with those specified by the Byte-IO WG.*

[4]At the time of writing we became aware of a new WG that aims to look at APIs for data movement - 'Data Movement Interface Standardisation' Working Group

The DAIS-WG is dealing with access to remote data, but focuses mostly on Data Bases. That topic is outside the scope of the SAGA API for now. However, the development of both groups should be cross-checked now and then for potential overlap.

- *The DAIS-WG offers no direct overlap with the SAGA-WG as of now, but future developments of the groups should continue to monitor each other.*

Logical Files

Several groups in GGF cover/covered the area of file replica management in Grids: REP-RG, OREP-WG, PA-RG, and CIM-WG.

The PA-RG has published a document (GFD-26 [7]) which lists the most common paradigms and operations in data Grids, in particular with respect to replica management.

- *The SAGA API spec should make use of the paradigms listed in the Persistent Archive RG document GFD-26.*

The listed GGF groups are mostly concerned with the definitions of a server architectures, and as such provide no immediate overlap with the SAGA API spec, at least not further as the PA-RG document cited above.

Streams

Although communication between remote entities is a central topic in distributed systems (and hence also in Grid systems), no groups apart from the GridFTP-WG touches that specific area. GridFTP has the primary focus of access to remote files, however, the FTP protocol by design can also be used as a memory to memory data transfer protocol, and hence, to a limited extend, for streaming of data. However, for the moment there is no obvious group within GGF which SAGA should be synchronize with in respect to data streaming.

- *There seems currently no need to synchronize a streaming API with other GGF efforts.*

Monitoring and Steering

As far as we know, there is no ongoing GGF work that focusses on *application* monitoring and steering (as apposed to job and resource monitoring and control, which is well covered). However, WSRF and related middleware specs do provide means for message exchange and notification, and the SAGA group

should, as already stated above, check the paradigms provided by these specs for compatibility.

- *There seems currently no need to synchronize an application level steering and monitoring API with other GGF efforts, however, lower level groups should be consulted with respect to available paradigms.*

Other Areas

The SAGA WG has been approached by, and probably will continue to draw interest from, other groups in GGF, which find the idea of both a high level application oriented API, and of a common look and feel over a range of Grid APIs very promising. Amongst these groups are notably GridRPC and GridCPR, which both are actively interacting with the SAGA group about extension of the SAGA scope into their domain. The GridRPC group submitted a number of use cases to the SAGA group for consideration.

Based on those use cases, the SAGA-RG is currently forming a dedicated design team to explore the common space of SAGA and GridRPC. That seems promising, and to be the right approach.

Other groups in GGF might in the near or far future consider a similar approach of integration and collaboration with the SAGA-RG (the GridCPR WG beeing a notable example).

- *The currently well defined scope fits the SAGA-WG work practices very well. Future scope extension should imply a revisit of the SAGA group structure and work model. The formation of dedicated additional design teams should be considered.*

Community Areas

The groups and members of the GGF community areas are ultimately representing the customers of the SAGA-API. The current driving force (the SAGA use cases) and feedback loop (SAGA use case authors and active group members) encompass only a small subset of the GGF community area. If that link is not strengthened, the SAGA API will be in danger to miss the target scope. Both feedback and additional use cases should be seeked for in the GGF community area (and elsewhere as appropriate) as soon and as frequently as possible.

- *The SAGA-WG should as early as possible search for feedback from the GGF community areas.*

4.2 Relation to Major Grid Middleware

It seems obvious that the SAGA API should appeal (functionally) to those application programmers who, prior to SAGAs availability, programmed against the currently available and widely deployed Grid Middleware, such as Globus, Unicore and gLite. Although a certain additional level of abstraction is to be expected by intend and design, care should be taken of compatibility with (and implementability of) the programming paradigms available in this Grid systems.

- *The programming paradigms exposed by the SAGA-API should be checked for compatibility with the paradigms offered by existing Grid middleware systems, and tools.*

4.3 Language binding Requirements

The SAGA-WG stated its intend to define the SAGA API in a language independent way. Hence the group leaves a certain amount of flexibility to various language bindings (which are eventually also to be produced in the SAGA-WG), e.g. to allow for language specific models for error handling, asynchronous operations, or for the handling of multiple output parameters. This section tries to list some high level requirements in respect to future language bindings.

The submitted use cases list a number of languages as target for the SAGA API: C++, C, Java, Fortran, Perl, and Python. Most of these languages are object oriented, C and Fortran are not. Earlier experiences in the group (e.g. with the GAT) have shown, that a mapping of OO API specifications to non-OO languages is possible, and actually straight forward. On the other hand it is a well accepted fact that mappings of a procedural API specifications to OO languages often end up not to be OO, but, well, procedural. As such it seems advisable to define the SAGA API in OO fashion, as that seems to facilitate the process of defining language bindings.

- *The SAGA API specification should be Object Oriented.*

Central to most languages are means of error handling, parameter handling, consistency and concurrency. Those issues however will also be touched by the SAGA API specification. Previous experience and the opinion of group members shows, that the acceptance of an API increases significantly if it feels 'native', i.e., if it adheres to the native paradigms of the language for error handling etc.

- *Language bindings should be able to replace SAGA API features with language native means if those are available and widely accepted, and if those do semantically support the intent of the SAGA API specification.*

- *The language binding definitions should strive to provide common look and feel and, most importantly, semantics for the SAGA API across all languages.*

4.4 Portability

Portability can be an issue at all levels: for API specifiers, implementors as well the API consumers and it might just be a question of where to park the portability problem. We believe it should be parked in the scope of the implementors whenever possible. In general, the API should be implementable on any platform using any programming paradigm.

Language specific bindings (procedural or otherwise) will be more sensitive to portability issues than the language independent API specification. For example one could argue that, as the scope of the API is limited, there is no real need for templates – but if used, the APIs portability is an issue for several specific language binding.

Although portability is definitely an issue for API specifiers, the overriding concern for the specifiers should be *simplicity of the API*. Not to imply that the specifiers are to maliciously make things non-portable, but when faced with the choice of keeping the API simple or possibly more generic the tilt should invariably be towards the former. For example, it is felt that there is a strong need to provide a simple mechanism for asynchronicity. The implementation details of asynchronicity in addition to being non-trivial, also varies significantly across platforms (and not just languages, see above).

- *For the SAGA API specification, simplicity of use should be more important than simplicity of implementation.*

5 Summary

This document derives the requirements – functional and non-functional – to help define the scope and design of the SAGA API. The resources used to derive the requirements are the use cases submitted to the SAGA-RG group, feedback and documentation from projects with similar scope as the SAGA API, as well as from efforts by other GGF groups on APIs (e.g. DRMAA) and infrastructure.

SAGA should support a simple, minimally complete, consistent, uniform set of function calls and thus be widely usable. Defining and narrowing the scope of the first version of SAGA to a level that would make it widely usable and quickly implementable, whilst ensuring its stablilty to future changes is both critically important and challenging – it requires balancing available expertise with community demand.

The main conclusions in terms of requirements are that the SAGA API should

support, at the minimum, the following functional areas: *job and resource management, resource discovery, data access, data management (including logical files)*, and *information services* (for persistent storage and retrieval of application level information). Also, given the strong requirements presented, *streaming* and*remote procedure calls* are also determined to be functional areas that are appropriately placed in the SAGA API scope.

Similarly, the non-functional areas that should be considered when designing the SAGA API specification, are deemed to be bulk operations, error handling, and, importantly, asynchronous operations.

The priority with which these requirements are addressed by the SAGA RG will depend on a several factors – such as available expertise within the SAGA group, contributions from other GGF groups, available resources etc. In fact, the current SAGA Strawman API does not cover all the requirements discusses in this document for these and other reasons. It was deemed appropriate to complete a subset of listed recommendations in the SAGA RG and address others as time and resources permit, rather than having an extended Strawman API scope and consequently a delayed version 1.0 of the SAGA API.

6 Security Considerations

This document is informational, and contains a set of use cases. As such, it does not address security considerations directly. Security, however, is discussed in this document, and several security requirements to Grid APIs are explicitely listed.

7 Contributors

Most significant acknowledgments are to be given to the many use case contributors, without whom, this analysis would not have been possible. We thank members of the SAGA Design team and many others for providing useful feedback and discussion. We thank Tom Goodale for setting up an initial structure of this document. The authors acknowledge useful discussions with Thilo Kielmann regarding the use of RFC2119. We thank Graeme Pound and Steven Newhouse (OMII) for useful feedback and for informing us about the existence of the Data Movement Interface Standardisation Working Group.

8 Intellectual Property Statement

The GGF takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation

or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the GGF Secretariat.

The GGF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights which may cover technology that may be required to practice this recommendation. Please address the information to the GGF Executive Director.

9 Disclaimer

This document and the information contained herein is provided on an As Is basis and the GGF disclaims all warranties, express or implied, including but not limited to any warranty that the use of the information herein will not infringe any rights or any implied warranties of merchantability or fitness for a particular purpose.

10 Full Copyright Notice

References

[1] Gabrielle Allen, Kelly Davis, Tom Goodale, Andrei Hutanu, Hartmut Kaiser, Thilo Kielmann, André Merzky, Rob van Nieuwpoort, Alexander

Reinefeld, Florian Schintke, Thorsten Schütt, Ed Seidel, and Brygg Ullmer. The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid. *Proceedings of the IEEE*, 93(8):534–550, 2005.

[2] Rüdiger Berlich, Marcel Kunze, and Kilian Schwarz. Grid computing in Europe: from Research to Deployment. In *CRPIT '44: Proceedings of the 2005 Australasian workshop on Grid computing and e-research*, pages 21–27, Darlinghurst, Australia, 2005. Australian Computer Society, Inc.

[3] S. Bradner. RFC 2119: Key words for use in RFCs to Indicate Requirement Levels. `http://www.ietf.org/rfc/rfc2119.txt`, 1997.

[4] Global Grid Forum. GGF Use Case Repository. `http://www.ggf.org/ucr/`.

[5] Global Grid Forum. Distributed Resource Management API Working Group, 2003. `http://forge.gridforum.org/projects/drmaa-wg/`.

[6] Andre Merzky and Shantenu Jha. Simple API for Grid Applications – Use Case Document. Technical report, Global Grid Forum, March 2006. GFD.70.

[7] Reagan Moore and Andre Merzky. Persistent Archive Concepts. Technical report, Global Grid Forum, December 2003. GFD.26.

[8] Naregi, Japan. The Naregi Project. `http://www.naregi.org`.

[9] Hrabri Rajic, Roger Brobst, Waiman Chan, Fritz Ferstl, Jeff Gardiner, Andreas Haas, Bill Nitzberg, Hrabri Rajic, and John Tollefsrud. Distributed Resource Management Application API Specification 1.0. Technical report, Global Grid Forum, June 2004. GFD.022.

[10] Raul Sirvent, Andre Merzky, Rosa M. Badia, and Thilo Kielmann. GRID superscalar and SAGA: forming a high-level and platform-independent Grid programming environment. In *CoreGRID Integration WorkShop 2005*, 2005.

[11] University of Barcelona, Spain. Grid SuperScalar. `http://www.bsc.es/grid/`.