GFD-E.176
SAGA-WG

Mathijs den Burger, VU
Manuel Franceschini, VU
Malcolm Illingworth, EPCC
Ceriel Jacobs, VU
Shantenu Jha, LSU
Hartmut Kaiser, LSU
Thilo Kielmann, VU
Andre Merzky[1], LSU
Rob van Nieuwpoort, VU
Sylvain Reynaud, IN2P3
Ole Weidner, LSU

Version: 1.0

February 7, 2011

Experiences with Implementing the SAGA Core API

Status of This Document

This document provides information to the grid community, documenting implementation experiences for the 'Simple API For Grid Applications' (SAGA) as specified in GFD.90 [5]. It is supposed to inform implementors, and to support the process of defining SAGA language bindings. Distribution is unlimited.

Abstract

The SAGA Core API (or short, SAGA API) has been implemented by a variety of groups, in different languages. As the SAGA API specification itself is language neutral (it specifies the API in SIDL), it is difficult to define interoperability between these implementations, in the conventional sense. That is left to later experience reports addressing specific language bindings.

This report rather will show that (a) the SAGA API can be mapped to various programming languages, without losing any functionality, and (b) that these implementations can provide the required semantics for a wide variety of grid (and non-grid) backends. We consider those properties as necessary and sufficient to promote the proposed SAGA API specification (P-REC) to full recommendation status (REC).

[1] editor

Contents

1 Introduction

1.1 Purpose of this Document

The purpose of this experimental document is to document the experience gathered while implementing GFD.90 [5], the proposed recommendation for a 'Simple API for Grid Applications'. In particular, these experiences are to prove that interoperable implementations of the SAGA Core API specification are available.

1.2 Notational Conventions

In structure, notation and conventions, this document follows those of the SAGA Core API specification [5], unless noted otherwise. This document refers to [5] as 'SAGA Core Specification', 'SAGA Specification', or simply 'specification'. It further refers to the specification with applied errata (see appendix A) as 'Revised Specification' or simply 'Revision'.

1.3 Interoperability Metrics

The OGF document process in GFD.1 [3] says:

> *"Once a document is published as a GFD-R-P, a 24-month timer will begin during which period it is expected that operational experience will be gained will mean [sic] that at least two interoperable implementations (from different code bases and, in the case of licensed code, from two separate license agreements) must be demonstrated (if appropriate). The entire protocol or specification must be implemented in the interoperable implementations. The GFSG will determine whether interoperable implementations (or implementations in software at all) are necessary or whether operational experience can be gained in a more appropriate fashion."*

GFD.1 is, however, silent about the means to be applied while proving interoperability for the documented implementations. While interoperability is rather well defined for protocols and service interfaces, it is less stringently defined for APIs. Further, as SAGA is a language independent API specification, it is not immediately obvious how interoperability can be demonstrated for implementations in different renderings, i.e. in different programming languages.

We thus define interoperability below as it is applied in the remainder of the document. This definition focuses on two main aspects: implementability (language independent), and implementation-interoperability (language depended).

Implementability of GFD.90

GFD.90 is considered to be implementable, iff

A-1: multiple diverse language bindings can be defined which are able to express the complete semantic and syntactic scope of the SAGA Core API, while not compromising SAGA's major design goals, as outlined in GFD.90, in section 2; and

A-2: implementations of these language bindings exist, which cover the complete scope of GFD.90.

Note that this point does *not* imply that a specific implementation is able to switch between different language bindings on compile, link or runtime, but rather that semantically identical applications can be implemented in the various language bindings.

At the point of writing of this document, 3 different language bindings exist (C++, Java, Python), for which multiple implementations exist which cover the complete SAGA API scope. Although those language bindings are not yet standardised, they prove that bindings are possible and functional.

Interoperability of GFD.90 Implementations

GFD.90 implementations *in one specific language binding* are considered interoperable, iff

B-1: these implementations can interchangeably be used to execute the semantically same set of grid operations.

A stronger version of this requirement could be phrased as

C-1: GFD.90 implementations in a single language are considered interoperable if they use the same set of language-native API definitions[1].

This paper documents C-1 for some languages (Java and Python), which thus immediately proves B-1 for those languages. Note that it is, however, *not* the

[1] such as interface classes for Java, header files for C and C++, and abstract classes for Python

purpose of this document to define, or even document, GFD.90 language bindings (although it will show and discuss various features of these language bindings for illustrative purposes).

It is very important to understand that this document, and indeed the SAGA API specification, is silent about protocol-level interoperability, and adaptor-level interoperability (see Section 2): as important as those aspects are, they are out of scope for the present discussion, and in fact out of scope for SAGA as an API specification.

1.3.1 Goals of this Document

In accordance with the implementability and interoperability metrics defined above, this text documents

D-1: the existence of semantically complete language bindings in C++, Java and Python,

D-2: the API-level interoperability of two implementations of the Java language bindings (which use the same Java interface classes), and of two implementations of the Python language bindings (which use the same python classes to interface to different implementations).

We consider those experiences sufficient to document the implementability of GFD.90.

Note that we do not attempt to present API compliance tests: we consider those to be useful and required for specific language bindings of the SAGA API – they seem, however, neither appropriate nor applicable to a language independent API specification. The various implementation usually do implement extensive unit tests, which we intent to promote to compliance test suites for the various language bindings.

1.4 Structure of this Document

The following sections will will document the various known implementations of GFD.90, grouped by implementation languages. For each implementation language, major language bindings design issues will be discussed and motivated. The document will then discuss the individual metrics defined above, and document experience and evidence which supports each one of them individually.

2 Implementation Experiences

This chapter discusses the implementation experiences for GFD.90 by describing the following 6 SAGA implementations[2]:

1. **C++**
   (a) SAGA-C++ (LSU)
2. **Java**
   (a) DESHL (DEISA, EPCC)
   (b) JavaSAGA (VU)
   (c) JSAGA (IN2P3)
3. **Python**
   (a) Python wrapper for C++ (LSU)
   (b) Python wrapper for Java (VU)

The implementation of GFD.90 revealed a variety of problems, from typos, over semantic inconsistencies, to some items which were simply not implementable in the form prescribed. Almost all of these problems are explicitely listed in the appendix, which is the source for the accompanying revision of GFD.90. The descriptions below will discuss some, but not all of these problems, mostly due to space constraints.

At time of this writing, two Python implementations of the SAGA Core API Specification exist. Both are implemented as wrappers, around the JavaSAGA and the C++ SAGA implementations, respectively, using the standard wrapping solutions Jython and Boost-Python. For historical reasons, these bindings *differ*[3]. We describe these bindings here anyway, as their existence proves that Python language bindings can in fact be sensibly defined.

2.1 Implementation Properties

The scope of the SAGA API as defined in [5] will most likely convince the reader that there don't exist many middleware systems, if any, that can provide the complete semantic set of functionality required to implement the SAGA API. Further, the extensibility of the API suggests that a single middleware system can certainly not be expected to cater for future API extensions.

[2] alphabetically ordered, grouped by language binding

[3] There is an ongoing effort at the VU, Amsterdam, to unify the two python language bindings. That effort is not advanced enough to be included here.

Any complete SAGA implementation will, in practice, need to bind against multiple middleware implementations (backends).

Further, as different middleware backends are likely to overlap in their functionality, SAGA implementations will likely be able to switch between backends for the same functionality, possibly at runtime. This will support application portability, which is a declared goal of the SAGA effort.

Many SAGA implementation will, in practice, allow switching between different backends, at compile, link, and/or runtime.

One widely accepted design pattern to implement the above properties is the *adaptor pattern*: small, self contained pieces of code (adaptors) translate requests from the upper layers (the SAGA API) into requests to the lower layers (the individual backends) – see figure 1.

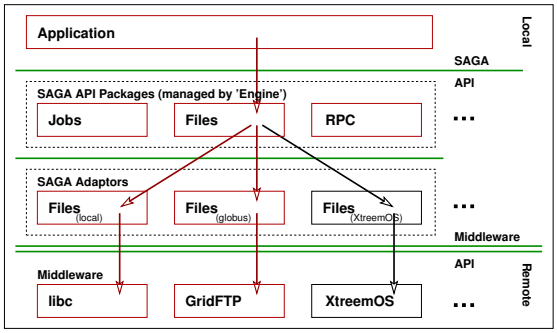

Figure 1: **Abstract architecture of SAGA implementations**

In fact, from the six individual SAGA implementations we discuss in this document, three are adaptor based implementations and two are wrappers around adaptor based implementations. Even the sixth implementation follows that general design principle, and includes a rather self contained abstraction layer which interfaces the SAGA implementation to its (only) backend.

It seems prudent to assume that SAGA implementations will often be adaptor based.

It should be noted that the SAGA API specification in no way prescribes these properties, nor does it prescribe any specific implementation architecture – apparently those properties simply emerge from the boundary conditions defined by the API's scope and structure.

2.2 C++

At the moment, there exists one SAGA implementation in C++, called SAGA-C++, which implicitely defines the C++ language bindings of SAGA. As the SAGA Core API specification is object oriented, the mapping to the C++ language is straight forward, and can almost always be directly derived from the IDL specification in GFD.90.

The use of C++-templates has been kept to a minimum, and has only been used in those cases where explicit type conversions are required on API level, e.g. on `task.get_result <type> ()`.

A major point of discussion for the language bindings has been the rendering of asynchronous object construction, the access to the default SAGA session instance, and the rendering of the SAGA task model. Those items have been rendered in ways which seem consistent with similar renderings in other widely used C++ libraries, and are thus considered to be successfully mapped into the C++ language.

As a side note: SAGA-C++ follows the design principles discussed in the previous section to the letter: it is adaptor based, has late-binding, and is extensible. Its portability and BSD like license makes it usable in a wide variety of application environments. As the initial investment in implementing SAGA is very high, it seems thus unlikely at the moment that a second C++ implementation will be forthcoming anytime soon. That will prove to be a problem for finalizing the C++ language binding specification process – which is not part of this document though.

2.3 SAGA-C++

2.3.1 Overview

The SAGA C++ implementation, called SAGA-C++, originates from within the SAGA Research and Working group in OGF – the implementation group at CCT/LSU is driving both the specification of the C++ language bindings, and the evolution and convergence of the SAGA Core API Specification.

The implementation work has been funded from a wide variety of resources: CCT internal funds, several projects funds such as XtreemOS (EU), the LONI Institute, two OMII-UK projects, NSF and TeraGrid funding, etc. Additional to a core of about three developers, several students, GSOC[4] students, and several external developers contributed to the code base. The work fostered a number of academic publications, centering on application level grid interoperability, programming patterns for distributed applications, grid/cloud interoperation, and software engineering aspects of the SAGA implementation. The implementation, along with its Python binding (see section 2.9) is used by a number of projects, mostly in the US, UK and Germany.

For additional material (documentation, tutorial slides, test suite results, mailing list, download links, etc.), see `http://saga.cct.lsu.edu/`.

2.3.2 Implementation Scope

SAGA-C++ implementation is a complete SAGA compliant implementation: it covers all functional and non-functional packages of the SAGA Core API specification.

As most SAGA implementations, an adaptor based late binding architecture was chosen to provide seamless runtime portability between different backends. Backend bindings exist for a wide variety of systems, such as (implemented packages in brackets):

- local systems: Unix/POSIX, MacOS, Windows (all packages)
- globus (context, job, file, replica)
- condor (context, job)
- ssh (context, job, file)
- clouds: ec2, eucalyptus, nimbus (context, job), opencloud (sector/sphere: context, job, file)
- lsf (context, job)

[4] Google Summer of Code, sponsored by OMII-UK

- glite (context, job)
- hbase (file)
- hdfs (file)
- htbl (file)
- kfs (file)
- ninfg (rpc)

Some adaptors have been implemented, funded or supported by external groups. An adaptor generator exists which simplifies the development of adaptors significantly.

Noteworthy is that an ever increasing set of unit tests exists, which is ensuring (i) the syntactic correctness of the C++ API, (ii) the semantic correctness of the C++ implementation, and (iii) the semantic equivalence of the various middleware bindings of that implementation. At the same time, several aspects of application-level interoperability have been demonstrated by SAGA-C++ based projects, by (i) using SAGA-C++ to coordinate distributed application components; and (ii) by using different adaptor sets within one application for semantically equivalent operations.

In terms of engineering, boost has been chosen as the main C++ toolkit to support the implementation work. All boost versions starting from 1.33.1 are supported. Boost is the only external dependency of the implementation – all other dependencies are introduced on adaptor level (globus, ...).

As the SAGA specification work continues within OGF, the implementation has also been used to prototype and implement SAGA extensions, such as Service Discovery, Adverts, CPR, and Information Services. Adaptors to these experimental packages exist, some are part of the regular release package.

2.3.3 Implementation experiences (Language independent)

As both the SAGA-C++ implementation from CCT and the JavaSAGA implementation from VU have been very early implementations, which started well before the SAGA API became an OGF proposed recommendation, the encountered issues are numerous and diverse. This section will only discuss a sample of those issues, selected by their significance in respect to specification evolution and language binding definition (see next section). Basically all issues have been resolved in the SAGA Revised Specification or in the implementation.

The C++ implementation of SAGA includes a proof-of-concept implementation of the SAGA Bulk optimization as described in the Core specification. SAGA Bulk operations are defined as a collection of asynchronous (task) operations

which are managed in a single task container. The C++ implementation could demonstrate that this is actually implementable, and performant.

The SAGA-C++ implementation motivated the inclusion of a SAGA URL class into the Core Specification, to provide a more uniform semantic to URL interpretation. In particular with respect to late binding, and the related need for URL translation for different backends, that additional URL class is very beneficial.

A number of job description attributes have been identified as missing from the Core Specification, and have been added to the Revised Specification since. Amongst them are the JSDL SPMD attributes, JobProject, and others.

Many SAGA operations have default flags specified. In many cases, however, these default flags turned out to be impractical, contradictory, or simply annoying. That has been rectified throughout the Revised Specification.

Semantic clarifications have been motivated for the Revised Specification, for many API calls, such as `migrate()`, `file.move()`, `session.add_context()`, and others. Although the SAGA API Specification is rather specific in specifying method semantics, inter-method semantics turned out to be contradictory or cumbersome in several places – that has been addressed since.

2.3.4 Implementation experiences (Language dependent)

As SAGA-C++ is as of now the only C++ implementation of the SAGA Specification, it in some ways defines the (draft) C++ language bindings. Most language specific issued we encountered can thus be regarded as language binding issues instead of real SAGA Specification issues.

The most prominent item is certainly the rendering of the SAGA task model in C++. Although the current rendering is rather elegant (it follows quite closely to what is described in the C++ based examples of the SAGA Core Specification), it left a large number of syntactic and semantic details to be fixed. This is however considered to be successfully solved, and the asynchronous SAGA operations are complete and consistently implemented.

Further, as discussed in the errata appendix in section A, the exception based error reporting mechanism originally defined by the SAGA Specification turned out to be impossible to implement, and needed some relaxation.

Another major challenge was the semantic clarification of the implications of the late binding architecture which is used by most SAGA implementations to date. Late binding has in particular implications for error reporting (see above), object state, object lifetime, and parameter interpretation. Especially URL handling

and interpretation is challenging in late binding implementations. Many small and large clarifications have been added to the Revised Specification, based on the C++ implementation experiences.

SAGA-C++ leans heavily on the boost [2] libraries. A small set of boost classes are actually exposed on API level. It is not the purpose of this document to justify or discuss that SAGA API dependency on boost – that is left to the C++ language binding document.

In general, the (Revised) Specification and the C++ bindings converged quite smoothly, and no outstanding issues with the specification are known which need to be addressed in the current version of the SAGA API.

2.3.5 Summary

We consider the SAGA-C++ implementation to be a complete and compliant SAGA API implementation. Numerous experiments in a variety of applications and projects [12, 10, 4, 9, 6, 7, 8, 13] have shown the ability of the implementation to provide application-level interoperation and portability on both semantic and syntactic level. The SAGA extension mechanism (stable Look-&-Feel, additional packages, late-binding implementations) has proven to be very useful and ensures a continuous evolution of the SAGA landscape.

2.4 Java

In most repects, the Java language bindings are, just as the C++ bindings, a straight forward mapping from the GFD.90 IDL specification to the Java language. There are a number of SAGA API elements, however, which required some deviation from the IDL in order to achieve a native look and feel.

In order to cater to the Java typical object live cycle, object creation is done via class factories. That approach elegantly solves the problem of asynchronous object creations.

The SAGA task model has been rendered slightly different from C++: task instances are not typeless as in C++, but are typed according to the return values of the method call the task represents.

The SAGA File API focuses on random I/O. While that is available in Java, the dominating file I/O paradigm is doubtless streaming I/O. The Java language bindings are thus adding streaming I/O to the file package.

Apart from those points, the Java langage bindings are sticking to the object oriented syntax as defined in the GFD.90 IDL. The language bindings are implemented as a set of abstract Java interface classes, which are then implemented by the SAGA implementations. It should be noted that the JavaSAGA and the JSAGA implementations use the same set of Java interface classes, and are thus by definition interchangably usable by any SAGA application written in Java.

2.5 DESHL

2.5.1 Overview

DEISA is a European consortium of leading national supercomputing centers that deploy and operate a persistent, production quality, distributed supercomputing environment. DEISA is, at its core, a grid of HPC resources. The purpose of this FP6-funded EU research infrastructure is to enable scientific discovery across a broad spectrum of science and technology, by enhancing and reinforcing European capabilities in the area of high-performance computing.

Amongst the DEISA objectives are (i) user transparency (users should not be aware of complex grid technologies); and (ii) applications transparency (minimal intrusion on applications, which, being part of the corporate wealth of research organizations, should not be strongly tied to an IT infrastructure). Toward those objectives, the JRA-7 group of the DEISA project designed DESHL[5], a single user interface for accessing OGSA-based services for distributed resources, which integrates several existing user-level tools to provide high-level services for:

- authentication, authorization and accounting;
- job preparation, submission and monitoring;
- data movement for job input and output;
- other areas as determined by DEISA user requirements.

In order to keep DESHL independent from the underlying heterogeneous infrastructure, JRA-7 decided to add a SAGA layer between the DESHL command-line client and the lower level grid access libraries. That additional layer is a partial SAGA implementation, and this section describes its implementation experiences.

2.5.2 Implementation Scope

The "DEISA Services for the Heterogeneous Layer" (DESHL) SAGA library covers only the SAGA job and file packages, and the SAGA Look-&-Feel, the semantics for the other SAGA packages was at that time not provided by the middleware DESHL was targeting (Unicore-5) . DESHL is thus a *partial SAGA implementation*. It binds to the DEISA infrastructure, via the ARCON client library which interfaces to Unicore.

As DESHL was designed and implemented while the SAGA implementation was still in flux, and not published, it did not follow all API draft changes, at least

[5]DEISA Services for the Heterogeneous management Layer

not immediately. The early implementation experience did, however, influence the evolution of the SAGA specification, also after it got published, and the description of DESHL is thus included in this document, even if DESHL is, strictly spoken, not fully SAGA compliant.

A significant motivation to use SAGA for the DESHL API was to shield the DESHL developers from future changes to the DEISA infrastructure, e.g. the planned transition from Unicore-5 to Unicore-6 – even at that early age of SAGA, infrastructure changes were deemed more disruptive than specification evolution. It has been found that was possible to port DESHL from Unicore-5 to Unicore-6 leaving the actual application code largely unchanged.

2.5.3 Implementation experiences (Language independent)

The SAGA Job and File packages mapped rather well to the required functionality of the DESHL command line utility. The implementation led to number of changes to the SAGA API being proposed, and ultimately accepted, such as the support for job I/O staging, the closer adherence to the JSDL specification, and the generalization of the SAGA context semantics (which was too limiting for the DEISA use case).

2.5.4 Implementation experiences (Language dependent)

The actual rendering of the language independent SAGA specification into Java language bindings was relatively straight forward, but differed at several points from the nowadays more widely used SAGA Java bindings from the VU Amsterdam (which will most likely form the basis of a future Java language binding specification in OGF). Nevertheless, key design elements, such as the use of factories for object creation, or the syntax for attribute settings, are the same in both renderings.

2.5.5 Summary

Despite of the differences between the DESHL implementation and the future SAGA API bindings for Java, which can be traced to the early start of DESHL, we can confidently claim that in our experience, the SAGA API is implementable in Java, and that the implementation provides the syntactic and semantic scope of the SAGA API, as indented.

2.6 JavaSAGA

2.6.1 Overview

The JavaSAGA project consists of two parts: providing Java language bindings for the SAGA specification, and providing an implementation. JavaSAGA has been developed at the Computer Science department of the Vrije Universiteit, and was funded by OMII-UK. Funding was for one man-year. Of that man-year, about 10 weeks were spent creating the language bindings. The rest of it was used for the implementation.

The Java SAGA language bindings mostly follow the SAGA specification. Some exceptions were made, e.g. to follow the Java coding conventions for naming identifiers, and to use exceptions instead of POSIX error codes. More details are provided below.

The JavaSAGA implementation is adaptor-based. The implementation provides service provider interfaces (SPIs) that must be implemented by the adaptors. In addition, it provides base classes for the adaptors that may be used to ease programming. The implementation uses late binding. Due to Java, the implementation engine and most adaptors are completely operating-system independent. JavaSAGA is also self-contained, i.e. it runs out of the box. All dependencies are included (except, of course, user-specific credentials, such as globus certificate files or ssh key files).

Additional material (documentation, download links, etc.) is available at [1].

2.6.2 Implementation Scope

JavaSAGA provides implementations of all packages of the SAGA Core API specification. Bindings exist for a variety of systems:

- **Local systems**: namespace, file, job
  Available in any OS that has a JVM (e.g. Unix, MacOS, Windows, etc.)
- **generic**: replica
  Provides a replica system built on top of other SAGA packages.
- **Globus**: context, job, namespace, file
  Provides (through JavaGAT) support for Globus 2.x, Globus 3.x and Globus 4.0. A JavaGAT adaptor for Globus 4.2 is almost done.
- **SSH**: context, job, namespace, file
  There are two versions: one using your local ssh command, and one talking directly to an ssh server via the Trilead SSH library for Java.

- **GridSAM**: job
  JavaGAT has a GridSAM adaptor, but JavaSAGA also provides an adaptor that was built directly on top of the GridSAM client libraries.
- **gLite**: context, job, namespace, file
- **FTP**: context, file, namespace
- **SFTP**: context, file, namespace
- **TCP sockets**: stream
  Implemented on top of the class `java.net.Socket`
- **Apache XML-RPC**: rpc
- **Unicore**: job

2.6.3 Implementation experiences (Language independent)

For some packages, JavaSAGA only provides an adaptor that is built on top of the JavaGAT system. JavaGAT can be seen as a predecessor to SAGA. It is also adaptor-based, uses late binding, and has adaptors for Globus, gLite, ssh, and others. Since JavaGAT is not SAGA, some SAGA features could not be implemented. In the remainder of this section, we will list the limitations of the JavaSAGA implementation per SAGA package.

Stream package

The JavaGAT adaptor for the stream package is built on JavaGAT's `EndPoint` and `Pipe` classes. The adaptor implementation also depends on the JavaGAT `AdvertService`. The TCP adaptor for the stream package is built on the `java.net.Socket` class.

Both adaptors do not support the STREAM_WRITE metric because the underlying systems do not support it either. The same holds for the method `waitFor(Activity.WRITE)` and stream attributes (although the BLOCKING attribute may be supported in the future).

Job package

The JavaGAT adaptor for the job package is built on JavaGAT Resources, for which various adaptors are available. There is also a dedicated SAGA adaptor for GridSAM. Neither of these adaptors currently implements `Job.getSelf()`

because we do not know any generic method (yet) by which a Java application can steer itself. Furthermore, JavaGAT is missing the functionality to implement the following SAGA features:

- The job description attributes SPMDVARIATION, THREADSPERPROCESS, JOBCONTACT, and JOBSTARTTIME.
- Post-stage append and pre-stage append in the file transfer directives.
- The job attribute TERMSIG.
- The job metrics JOB_SIGNAL, JOB_CPUTIME, JOB_MEMORYUSE, JOB_VMEMORYUSE, and JOB_PERFORMANCE.
- The methods `signal()`, `checkpoint()`, and `migrate()`.

The mapping from a SAGA job description to a JavaGAT job description is mostly possible, as is the mapping from SAGA job methods to JavaGAT job methods. However, this does not mean that JavaGAT adaptors actually implement everything.

The following SAGA features could not be implemented on top of GridSAM:

- the job description attributes WORKINGDIRECTORY, INTERACTIVE, JOBCONTACT, and JOBSTARTTIME.
- post-stage append and pre-stage append.
- the job attribute TERMSIG.
- the job metrics JOB_SIGNAL, JOB_CPUTIME, JOB_MEMORYUSE, JOB_VMEMORYUSE, and JOB_PERFORMANCE.
- the methods `signal()`, `suspend()`, `resume()`, `checkpoint()`, and `migrate()`.

Namespace package

Both the JavaGAT and the Local adaptor support all SAGA namespace features, except for links and permissions (see below for more details).

File package

Both the JavaGAT and the Local adaptor did not implement the extended I/O methods, for which there is no support in JavaGAT nor Java's local file API. In addition, `readP()` and `writeP()` only inherit the default implementation in the base class of the adaptor, which translates the patterns into seeks and contiguous reads/writes. This approach is very generic, but probably also very slow. The JavaGAT adaptor use JavaGAT's `RandomAccessFile` underneath, when it is available for the middleware layer at hand. For other backends, `FileInputStream` or `FileOutputStream` are used. The Local adaptor uses the more efficient `java.nio.FileChannel` class underneath to provide fast access to local files.

LogicalFile package

JavaSAGA only contains a generic adaptor for logical files. In this adaptor, a logical file is simply a normal ASCII file that contains a list of URLs, one per line. The SAGA file and namespace packages are used to manipulate the contents of a logical file.

RPC package

JavaSAGA contains an XML-RPC adaptor, which completely implements SAGA's RPC package on top of the Apache XML-RPC client library. The only limitation of this adaptor is that at most one OUT parameter is supported.

2.6.4 Implementation experiences (language dependent)

The first hurdle to take was to provide Java SAGA language bindings while the SAGA specification was still taking shape. The design of the language bindings was guided by two rules:

1. Follow the Java Code conventions.
2. Keep as close as possible to the SAGA specification.

Below is a summary of decisions made in the Java SAGA language bindings. The Java language version used is the one provided in J2SE 5.0, which is widely available. It is also the first version that provides generics and enumerated types, which are used throughout the SAGA language bindings for Java. J2SE 5.0 also provides the `java.util.concurrent` package, which is used for the Java language bindings of SAGA tasks.

Java language bindings for SAGA

For facilitating both application writing and implementing SAGA, providing the Java language bindings in the form of directly usable files was important. Both interfaces and classes from the language-independent SAGA specification are therefore provided in the form of Java interfaces. Since interfaces do not have constructors, factories are provided to create SAGA objects. This setup requires a bootstrap mechanism for creating factory objects. The mechanism uses the `saga.factory` system property, to be set by the user to point to an implementation-specific meta-factory, which in turn has methods to create factories for all SAGA packages.

SAGA Object

Since `Object` is a predefined class in Java, we opted to name the SAGA base object `SagaObject`. The `ObjectType` class is not included in the Java language bindings, since Java has an `instanceof` operator.

File I/O and Java file streams

Earlier experience with JavaGAT has shown that having implementations of the Java streams `java.io.InputStream` and `java.io.OutputStream` is very much appreciated by Java application programmers, since these are the types on which most Java I/O is based. Therefore, it was decided to add specifications for `FileInputStream` and `FileOutputStream` to the file package. The `file` class from the SAGA specifications is also specified in the Java language bindings.

Files and error handling

The SAGA specification refers to POSIX error return codes for several methods. However, it is not to be expected that these will be available in existing Java grid middleware. Also, in Java, error conditions are supposed to be passed on by means of exceptions. We therefore decided that where the SAGA specifications refer to POSIX error codes, a `SagaIOException` is to be thrown in these cases. This may be less informative than a specific error code, but is more 'Java-like', and probably easier to implement on existing Java grid middleware.

Permissions and links

Currently, the Java language does not provide any building blocks for permissions and links (not even locally). Although all methods concerning links and permissions are specified in the Java language bindings, none of the adaptors in the JavaSAGA implementation actually support them. These methods throw a `NotImplemented` exception when they are invoked. Future Java versions may alleviate this problem somewhat.

Buffer

In Java, arrays have a size that can be examined at run-time. The buffer creation methods in the Java language bindings therefore specify either a `size`, in which case the buffer is implementation-managed, or a byte array, in which case the buffer is application-managed.

Error handling

The layout of the result of the method `getMessage()` in the `SagaException` class follows the usual Java convention instead of the SAGA specification. The result prescribed by the SAGA specification can be obtained by invoking the `toString()` method (as usual in Java).

A simple mechanism exists for storing and examining exceptions that may be thrown by adaptors in adaptor-based SAGA implementations. In such implementations, the top-level exception (the one highest up in the SAGA exception hierarchy) is not always the most informative one, and the implementation is not always capable of selecting the most informative exception. In these cases, the implementation may opt to add the individual exceptions as nested exceptions to the exception thrown. The language bindings make it possible to iterate over the nested exceptions (i.e. the `SagaException` class implements the interface `java.lang.Iterable`). This mechanism is now also incorporated in the SAGA specification.

Jobs

An important deviation from the language-independent SAGA specification is that the method `JobService.runJob()` is specified differently: the input, output and error stream OUT parameters are not specified here, since Java has no

OUT parameters. Unfortunately, their absence, according to the SAGA specifications, implies a non-interactive job. Since interactive jobs should still be supported, a boolean parameter is added here to specify whether the job is interactive. If interactive, the streams can be obtained from the Job using the methods `Job.getStdin()`, `Job.getStdout()`, and `Job.getStderr()`.

NSDirectory

A Java-specific extension in the package is that the `NSDirectory` interface extends `java.lang.Iterable`. Applications can thereby easily iterate over all the entries in a directory.

Permissions

The SAGA permission flags are specified as a Java enumeration class. Methods are included to combine them into integers, and to determine if they are set in an integer (in Java, enumerations cannot be treated as integers).

RPC

In contrast to the language-independent SAGA specification, the `Parameter` interface does not extend the `Buffer` interface. The motivation for this is that most Java language bindings for RPC systems use `java.lang.Object` as the type of parameters, which implies that at least some types are serialized automatically (e.g. via runtime inspection of the actual type). If each parameter would be a SAGA buffer, the type of the parameters would be limited to byte arrays.

Stream

Since Java programmers are used to the classes `java.io.InputStream` and `java.io.OutputStream`, a mechanism is provided to obtain such streams from a `Stream` object.

Tasks

An earlier version of the SAGA specification modeled the result of asynchronous operations as an additional OUT parameter. Java does not have OUT parameters, and the async and task versions already have a return value: the task object. The question arose where to leave the result values of tasks. This resulted in the decision to make the Task object itself a container for the result value, and to add a method to obtain the result. This decision has found its way back into the SAGA specification.

In addition, many small inconsistencies were discovered during the development of JavaSAGA. These have been fixed in the SAGA specification.

2.6.5 Summary

JavaSAGA implements all packages of the SAGA specification, and is therefore a complete SAGA compliant implementation. Various institutes and projects are already using and/or experimenting with JavaSAGA, including the neuGRID team at the University of the West of England in Bristol (UK), the Science & Technology Facilities Council (UK), CERN, and the XtreemOS project.

2.7 JSAGA

2.7.1 Overview

JSAGA is a Java implementation of the SAGA specification. It has been developed at the IN2P3 Computing Center (CC-IN2P3), in the context of a grid interoperability project (IGTMD) funded by the French National Research Agency (ANR). The main goal of JSAGA is to enable uniform access to existing production infrastructures, and namely to submit jobs to several heterogeneous infrastructures with a single description of these jobs.

JSAGA is adaptor based. Adaptor interfaces are designed to ease plugin development and to enable efficient usage of underlying APIs. These interfaces are service-oriented. Some of them are optional and are used for optimization purpose. Some of them offer several options to implement the same functionality with different approaches. JSAGA uses early binding to middleware.

The core engine and most adaptors are independent of the operating system, and they do not require any additional package to be installed. For example, one does not need to run JSAGA on a gLite User Interface machine in order to be able to use gLite middleware.

Besides implementing the SAGA specification, JSAGA is also using it to enable seamless job submission to existing grid infrastructures. It deals with grid infrastructures heterogeneity (e.g. network filtering rules, supported certificate authorities, commands and services available on execution sites), in order to run collections of jobs on several infrastructures efficiently and seamlessly.

For additional material (documentation, slides, mailing contacts, download links, etc.), see `http://grid.in2p3.fr/jsaga/`.

2.7.2 Implementation Scope

JSAGA aspires to be a "SAGA compliant partial implementation".

It does not implement the RPC and stream functional packages, because these features are not available on many existing production infrastructures. Hence, implementing these packages would not help us achieve our main goal, which is enabling uniform access to these very infrastructures.

The other packages of the SAGA specification are almost fully implemented. Current limitations are related to 'steerable' interface (used by class 'job_self' only) from Monitoring Look-&-Feel package, to 'checkpoint' and 'get_self' methods from Job functional package, and to scattered, pattern-based and extended

I/O methods from File functional package. All these methods currently throw a 'NotImplemented' exception for any back-end. They are not yet supported because they have not been requested by current users, but we plan to implement them for improving compliance to the SAGA specification.

The JSAGA implementation of the Job, Namespace, File and Replica functional packages, as well as the Context Look-&-Feel package, is based on adaptors, as described earlier. These adaptors support components from various grid middlewares: gLite, Globus Toolkit (pre-WS and WS), Unicore, Naregi. They also support more commonly used technologies, such as X509, HTTPS, SFTP, SSH, and other components such as SRB and iRODS.

2.7.3 Implementation experiences (Language independent)

Thanks to the accuracy and consistency of the SAGA specification, the JSAGA implementation experience went rather smoothly. The SAGA interfaces cover almost all of our needs, so we had to deviate from the specification for a few issues only.

Context Look-&-Feel package:

A security context sometimes needs to be initialized before it is used by a functional package, but this can not be done at any time because the result of initialization depends on which attributes are set. In order to enable on-demand context initialization, we added an implicit behavior to the method 'getAttribute' for attribute name 'UserID'; this method initializes the security context before returning the value of attribute 'UserID' as requested. The Revised Specification allows for initialization of the security context; this is done when adding the context to the session. Consequently, we will remove our specific behavior and implement the one specified in the upcoming Revised Specification document instead.

When several security context candidates are available for a given URL, JSAGA throws an exception instead of automatically trying to connect with each of them to find the right one to use. We made this choice because connecting with an unexpected security context could lead to problems that are painful to recover, such as creating files with unexpected owner, submitting jobs that will be allowed to run but not to store their result, locking accounts because of too many failed connection attempts. The description of the 'AuthenticationFailed' exception has been modified in the SAGA Revised Specification to take into account our solution, and we will use this exception instead of our current non-standard one.

Job functional package:

We added, on user request, a job metric for intermediate job states, such as state 'QUEUED'. This proposal for change has been rejected because it is not supported by all back-ends and because this feature can be supported through the 'job.state_detail' metric. Indeed, although this metric was added to provide backend specific state details, it could also be used to provide SAGA implementation specific state details. Hence, we will replace our current backend state details implementation (which is not used) with a JSAGA specific state details implementation, and then we will remove our non-standard job metric.

We made 'CPUArchitecture' and 'OperatingSystemType' job description attributes scalar in order to prevent the risk of information loss when converting a SAGA job description to a JSDL job description. This is fixed in the Revised Specification.

Among the widely used job description attributes, attribute 'Queue' is the only one that prevents to create a job description that can be submitted to several heterogeneous infrastructures. Indeed, there is no inter-grid convention for queues, neither for their naming, nor for the underlying semantic (although job lifetime is widely used). Since a queue is part of the resource location rather than part of the job description, we chose to encode it within the resource URL instead. This breaks application portability. Resolution of this issue will possibly be postponed to next JSDL version, or to a SAGA resource extension, whichever comes first. This change does not exactly violate the letter of the SAGA specification, but uses a semantic ambiguity.

We added, upon contributor's request, a read-only job attribute to dump the job description generated for the target backend. This attribute does not need to be included in the SAGA specification because it is used for debugging purpose only. The SAGA specification is silent about debugging extensions to the API.

File and Namespace functional packages:

We added, on user request, a method to the `namespace::entry` class in order to enable to get the last modification date of the entry. The Revised Specification includes a `get_mtime()` for that purpose, and we will rename our method to use that name.

For the `remove()` method of the `namespace::entry` class, we made the flag `Recursive` optional for empty directories, so that we can implement the behavior of the `rmdir` Unix command. This is fixed in the Revised Specification.

In order to fix a performance issue when reading or writing many small files with some protocols, we added a flag which enables to bypass file existence check when creating a new `namespace::entry` instance. Using a non-standard flag breaks application portability, in particular if other SAGA implementations throw exception on unknown flags, or if they use the same bit for their own non-standard flag. Since no agreement has been reached yet, this item will be

postponed to a later version of the SAGA specification. Until then, we will keep our non-standard flag in JSAGA, while warning users that it breaks application portability and proposing asynchronous calls as a (partial) alternative. This issue has been acknowledged by the SAGA group, and has been added as a TODO item for the next API revision[6].

2.7.4 Implementation experiences (Language dependent)

JSAGA implements the interfaces of the reference Java language binding of the SAGA specification. Since these interfaces are provided by the Vrije Universiteit, Amsterdam, we didn't have to cope with any language dependent issue. See section 2.6.4 in the JavaSAGA description for a detailed discussion on language dependent issues.

The reference Java language binding of SAGA enables using all the SAGA features without any dependency on the implementation classes (except the name of the implementation bootstrap). This binding takes advantage of Java 1.5 features, in particular the support for generics and type-safe enumeration.

The only issue encountered with this binding consisted in the need to patch it in order to provide our own implementation of the 'URL' class. Our motivation for this was to support automatic URL encoding, to support relative paths and to correctly manage local file URLs on Windows operating system. This issue is solved since version 1.0 release candidate 2 of the binding. Indeed, an 'URLFactory' class has been added to enable SAGA implementers to provide their own implementation of the 'URL' class without any patching.

2.7.5 Summary

JSAGA follows the SAGA API specification. It also implements the reference Java language binding interfaces in order to guarantee syntactic compliance and prevent any compilation error when replacing a Java-based SAGA implementation with another one. It aspires to be a "SAGA compliant partial implementation" for multiple backends. However, this goal has not been reached completely since some methods are not yet implemented.

JSAGA applications are diverse, and include an academic and industrial grid web portal (Elis@ by CS-SI), advanced job submission tools (like SimExplorer by the Complex Systems Institute, and JJS by CC-IN2P3), and a multi-protocol file browser (JUX by CC-IN2P3).

[6]The Revision does not attempt to fix this issue, as the resulting changes have, at least potentially, rather far reaching semantic implications for other SAGA API methods and classes.

In order to fulfill all the requirements of our users, we had to make only a few small deviations from the SAGA specification in our implementation. Almost the entire resulting mismatch is now disappearing through modification of the JSAGA implementation, the SAGA specification itself, or both. Only two deviations remain: the place for specifying the job resource queue name and the additional `namespace::entry` flag. These issues will be postponed to later versions of Open Grid Forum specifications, respectively JSDL/SAGA resource extension and SAGA.

The SAGA specification provides a high-level interface, which suits perfectly well for uniform access to heterogeneous middleware. It is simple to use, and allows for efficient use of the lower-level legacy interfaces without exposing the user to the complexity. Consequently, the JSAGA group recommends accepting GFD.90 as an OGF standard.

2.8 Python

Just as the C++ and Java Language bindings, the Python bindings strive to adhere to the SAGA syntax as defined in the GFD.90 IDL specification, while making sure that language specific syntactic elements are used to provide a language native look and feel to python programmers.

A notable different to the other bindings is the rendering of the task model, which is expressed by method flags. Other minor deviations, such as the all-uppercase spelling or enums, are simple consequences of python API conventions.

Both discussed Python implementations are based on the same set of Python API classes. SAGA applications in python can thus immediately switch between both implementations.

2.9 Python over C++

2.9.1 Overview

The C++ SAGA implementation described in section 2.3 includes a python binding, which is implemented by using Boost-Python. Additional components on wrapper level ensure that SAGA elements such as attributes, asynchronous operations, and file I/O are rendered in a pythonesque manner.

It may be interesting to know that a significant number of projects prefer the use of the python bindings over the C++ bindings, presumably for ease of prototyping and development, simple deployment, and cultural reasons.

2.9.2 Implementation Scope

As the discussed python bindings represent a thin wrapper around the C++ SAGA implementation, both architecture and semantics of operations is exactly as discussed in section 2.3. The same set of adaptors is available for C++ and Python, and thus the same middleware bindings are available.

2.9.3 Implementation experiences (Language independent)

As the development of the C++/Python wrapper occurred in parallel to the C++ implementation work, all comments made there apply here as well. There are no known issues which would need addressing from the SAGA Specification side at the moment.

It should be noted, that several projects use the C++ and python bindings in parallel. Due to the fact that one wraps the other implementation, interoperability may come as no big surprise. However, we would like to mention that the same developers using both bindings experienced no difficulties whatsoever when mapping code from one binding to the other, both in terms of syntax and, more importantly, semantics.

2.9.4 Implementation experiences (Language dependent)

Although boost-python is an acknowledged path to provide Python interfaces to C++ libraries and classes, the resulting Python API is not always considered to be 'native enough' to be 'real python'. Obviously this is a very subjective and gradual criterion, which is also partially addressed by additional implementation work on wrapper and python level. However, the issue remains that the exact

python binding definition is likely to change the syntax, if not the semantics, of the current C++/Python wrapper.

No other major issues are thus far known to limit the implementability of the SAGA Core API specification in Python.

2.9.5 Summary

By definition, or by design, the C++/Python wrapper is a just as functionally complete and compliant SAGA implementation as the C++ implementation. The wrapping for python is mainly an engineering issue, somewhat diluted though by cultural and language binding issues. The successful use of the C++/Python wrapper in numerous projects [11, 14, 4] underpins the implementability and usefulness of the wrapper, and documents, in our opinion, the implementability of the SAGA API Specification.

2.10 Python over Java

2.10.1 Overview

Python SAGA started as a master's project at the Computer Science department of the Vrije Universiteit Amsterdam. The aim of the project was to first create Python language bindings for SAGA, and then create an implementation of these language bindings on top of JavaSAGA. We nicknamed the language bindings 'PySAGA', and the implementation on top of JavaSAGA 'JySAGA'.

All semantic functionality of JySAGA is provided by the underlying JavaSAGA implementation. JySAGA only acts as a thin layer between the Python language bindings and the JavaSAGA implementation. JySAGA combines Python and Java by using Jython: a Python interpreter written in Java. Jython makes it possible to mix Python and Java code, which made it relatively easy to translate Python calls to their counterparts in JavaSAGA.

2.10.2 Implementation Scope

JySAGA implements all packages specified in the SAGA Core API on top of JavaSAGA. JySAGA can therefore access exactly the same middleware as JavaSAGA.

2.10.3 Implementation Experiences (Language Independent)

Since JySAGA only acts as a thin layer between Python and JavaSAGA, the development of JySAGA did not trigger any new issues when mapping SAGA calls to middleware. However, its development did trigger some bugs in the JavaSAGA implementation. These have all been fixed by now.

2.10.4 Implementation Experiences (Language Dependent)

The design of the Python language bindings for SAGA followed two rules:

1. Follow the Python style guide.
2. Keep as close as possible to the SAGA specification.

Various open issues have been discussed on the mailing list `saga-rg@ogf.org`. In the remainder of this section, we will list the most prominent issues and design choices.

Python language bindings for SAGA

The Python language bindings are provided as a set of Python modules. Each module corresponds to a SAGA package, and contains skeleton code for all classes of that package. The classes contain all methods but without an implementation. Each method is extensively documented using Doxygen strings. These can be automatically converted to browsable documentation of the language bindings.

A Python SAGA developer can simply copy all the module files and implement all skeleton classes. A set of generic unit tests is available to test the functionality of a Python SAGA implementation.

Naming

The following naming conventions were used:

- class names: CapitalizedWords (e.g. `JobService`)
- package and module names: lowercase (e.g. `file.py`)
- methods, parameters, and variables: lowercase_with_underscores (e.g. `Directory.open_dir()`)
- constants: UPPERCASE (e.g. `file.Flags.CREATE`)

These conventions stem from the Python style guide, with the exception of the convention for constants. The reason to use UPPERCASE for constants was that SAGA defines the Permission enums 'Exec' and 'None', but both 'exec' and 'None' are reserved keywords in Python. Using UPPERCASE ensured a consistent naming scheme for all constants without any keyword clashes.

Method Overloading

Methods are defined in Python by using the `def` keyword followed by the method name and the parameters in brackets. Since `def` also overwrites a previously defined method with the same name, method overloading (multiple methods with the same name but different parameter types) is not supported in Python. Although the same effect can be implemented explicitly by inspecting the parameter types at runtime, problems arise when SAGA specifies overloaded methods with completely different sets of parameter types. An example of an overloaded method is the `copy()` method from the `ns_entry` and `ns_directory` classes

in the namespace package, which has the same name but different parameters and semantics in both classes. The language bindings therefore specify different names for such methods (in this example, `NSEntry.copy_self()` and `NSDirectory.copy()`, respectively).

Multiple Return Values

A method in Python can return multiple variables from one method call. This language feature was used for methods in the SAGA specification that specify multiple OUT parameters. An example is the method `JobService.run_job()`, which returns both the job itself and three handles to stdin, stdout, and stderr.

Default Parameter Values

Python can specify default values for parameters in a method definition. Parameters with default values should come after parameters without default values, so Python can determine which value in a method call belongs to which parameter. Default values improve usability, but imply a specific parameter ordering. The order of method parameters in the Python language binding therefore differs sometimes from the order in the SAGA specification. Examples are the methods `file.read(size=-1, buffer=None)` and `file.write(buffer, size=-1)`. This parameter order was chosen because a data buffer is always needed for writing, but is optional for reading.

Asynchronous Methods and Tasks

To deal with asynchronous methods and the creation of tasks, a `tasktype` parameter has been added to all methods in subclasses of the `Async` class. This parameter is always the last one in a method signature and defaults to `TaskType.NORMAL`. Consequently, a method is executed synchronously unless the tasktype parameter is specified. The syntax of both synchronous and asynchronous method calls therefore remains consistent and does cause conflicts.

Getters and Setters

Using getter and setter methods is not regarded "real python". Many class variables on which getters and setters operate can therefore also be accessed

through so-called properties, which specify which methods Python should call when a class variable is read or written.

The reason for also specifying getters and setters is that all methods in subclasses of the `Async` interface can be executed both synchronously and asynchronously, including getters and setters. A property can only use one of these two versions. The Python language bindings therefore define both a getter/setter and a property to access each class variable. Using a property will call the synchronous version of a getter or setter, while the getters and setters themselves can be used for both versions.

Binary Data

Python 2.x does not have a `byte` type. It was therefore difficult to find a good equivalent for SAGA's `buffer` class, which encapsulates a sequence of bytes. Strings can be used instead, but prevent a user from supplying an application-managed buffer to a `read()` call.

The current solution in the language bindings is to use Python's `array` module with type 'b' (signed char). This deviates from the Python wrapper in the C++ SAGA implementation, which does not include the `buffer` class at all and only recognizes (immutable) strings as input to or output from methods that would normally use SAGA buffers. It remains an open issue whether the whole concept of SAGA buffers is "real python", and whether it is necessary to include then in the Python language bindings or not.

Python versions

The Python language bindings conform to Python version 2.2, which is widely used and supported.

Python 2.6 also supports abstract base classes, which could be used to provide the Python language bindings as a set of directly usable files instead of a set of skeleton classes.

Version 3.0 of Python (released in December 2008) is incompatible with the previous versions. It distinguishes separate types for text and binary data, as opposed to previous versions that used strings of Unicode or 8-bit characters. The new data types `byte` and `bytearray` are used to hold a single binary byte and a mutable buffer of binary data. The `bytearray` type could (also?) be used to implement SAGA buffers in Python.

It remains an open question which Python version should be required for the

Python language bindings. An extra limiting factor it that JySAGA depends on Jython, which currently only supports Python up to version 2.5.

2.10.5 Summary

The Python language bindings expose all the SAGA Core API classes in a Python-specific manner. JySAGA is SAGA compliant implementation in Python that implements these language bindings completely. A similar glue layer between the Python language bindings and the C++ SAGA implementation is currently being developed.

3 Conclusions

This document's intent is to prove that implementations of GFD.90 are interoperable, as required by GFD.1 in order to promote GFD.90 to a full OGF standard recommendation. For that purpose, we defined interoperability as a combination of "implementability (language independent), and implementation-interoperability (language dependent)" (see section 1.)

We have discussed different implementations of the SAGA Core API specification, and have demonstrated that

A-1: multiple diverse language bindings can be and have been defined which are able to express the complete semantic and syntactic scope of the SAGA Core API, while not compromising SAGA's major design goals, as outlined in GFD.90, in section 2; and

A-2: multiple implementations of these language bindings exist, which cover the complete scope of GFD.90.

Even if those language bindings are not yet fed into the OGF standardization pipeline, they exist and are in use[7].

We have further shown, that

B-1: these implementations can interchangeably be used to execute the semantically same set of grid operations.

In particular, we presented two completely independent implementations which adhere to the same language binding (Java). We have further demonstrated that a single python implementation can interface to different SAGA implementations in C++ and Java, thus providing the full scope of these implementations. Also, we demonstrated that the C++ implementation and the derived Python implementation can interchangeably be used to implement semantically interoperable software components.

We thus conclude that the SAGA Core API specification as defined in GFD.90 (with applied errata) can be interoperably implemented in the sense required by GFD.1, and recommend to advance GFD.90 to full OGF Recommendation.

[7] Also, one cannot sensibly expect standardized language bindings before the standardization of the language independent specification is completed.

4 Intellectual Property Issues

4.1 Contributors

This document is the result of the joint efforts of many contributors, and in particular implementors. The authors listed here and on the title page are those taking responsibility for the content of the document, and all errors. The editors (underlined) are committed to taking permanent stewardship for this document and can be contacted in the future for inquiries.

Mathijs den Burger
mathijs@cs.vu.nl
Vrije Universiteit (VU)
Dept. of Computer Science
De Boelelaan 1083
1081HV Amsterdam
The Netherlands

Manuel Franceschini
livewire@koltern.com
VU, The Netherlands

Malcolm Illingworth
m.illingworth@epcc.ed.ac.uk
EPCC
The University of Edinburgh
James Clerk Maxwell Building
Mayfield Road
EH9 3JZ Edinburgh, UK

Ceriel Jacobs
ceriel@cs.vu.nl
VU, The Netherlands

Shantenu Jha
sjha@cct.lsu.edu
Center for Computation and
Technology (CCT/LSU)
Louisiana State University
216 Johnston Hall
70803 Baton Rouge
Louisiana, USA

Hartmut Kaiser
hkaiser@cct.lsu.edu
CCT/LSU

Thilo Kielmann
kielmann@cs.vu.nl
VU, The Netherlands

Andre Merzky
andre@merzky.net
CCT/LSU

Rob van Nieuwpoort
rob@cs.vu.nl
VU, The Netherlands

Sylvain Reynaud
Sylvain.Reynaud@in2p3.fr
Centre de Calcul IN2P3/CNRS
Domaine scientifique de La Doua
43 bd du 11 Nov.1918
69622 Villeurbanne Cedex
France

Ole Weidner
oweidner@cct.lsu.edu
CCT/LSU

The authors would like to thank all contributors to the SAGA specification and implementations, and the members of the SAGA OGF groups, for feedback and support.

4.2 Intellectual Property Statement

The OGF 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 OGF Secretariat.

The OGF 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 OGF Executive Director.

4.3 Disclaimer

This document and the information contained herein is provided on an "As Is" basis and the OGF 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.

4.4 Full Copyright Notice

case the procedures for copyrights defined in the OGF Document process must be followed, or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by the OGF or its successors or assignees.

A Errata Discussion

As discussed in many places throughout the document, implementing the originally submitted GFD.90 required a number of changes and amendments. None of those required major structural changes to the API, nor did the scope of the API change in any way. The Revised Specification will be submitted to the OGF editor along with this document. This appendix lists and discusses the main changes between the original and the amended GFD.90. A large number of typos, spelling errors and grammatical errors have been corrected as well – those changes are not listed individually. The list is unsorted.

1. package `saga::file`, class `iovec`:

   The `set_offset` and `set_len_in` methods also throw BadParameter when "out of bounds":
   `size >= 0 && len_in + offset > size`

2. package `saga::file`, class `file`:

   The `read_v` and `write_v` methods also throw BadParameter when "out of bounds" if the above conditions for their `iovec`s apply. When no `len_in` is specified, the buffer size is used instead as `len_in`. If, in this case, `offset > 0` a BadParameter exception is thrown.

   Note: an exception is only thrown on the I/O methods, as otherwise, it would often not be possible to increase the offset on a buffer at all, as offset and `len_in` cannot be set together atomically:

   ```
   iovec iov (10);     // size = 10, len_in = 10, offset = 0
   iov.set_offset (3); // bang
   iov.set_len_in (7); // would create a valid state again
   ```

3. package `saga::file`, class `file`:

   The `iovec` constructor can also throw `NotImplemented`, as its base buffer can, and as all constructors should be able to. This has been more explicitely clarified for Look-&-Feel classes, too.

4. The default flag for file open should be `Read`. The `Create` flag should imply `Write`. The `CreateParents` flag should imply `Create`.

5. `CreateParents` semantics is now clarified:

   `f.mv ("file.txt", "newdir/thing", flags::CreateParents)`

   The above creates a new directory 'newdir', and renames 'file.txt' to 'newdir/thing'.

   `f.mv ("file.txt", "newdir/thing/", flags::CreateParents)`

   The above creates new directories 'newdir/thing', and renames 'file.txt' to 'newdir/thing/file.txt'.

So, if the target does not exist, all path element before the last slash are considered parent directories to be made.

6. "The callback classes can maintain state between initialization and successive invocations. The implementation MUST ensure that a callback is only called once at a time, so that no locking is necessary for the end user."

   But also, the callback may remove conditions to be called again, i.e. shut down the metric, read more than one message, etc. Implementations MUST be able to handle this. This is documented now.

7. URL expansion is now clarified:

   "Any valid URL can be returned on `get_url()`, but it SHOULD not contain `..` or `.` path elements, i.e. should have a normalized path element. The URL returned on `get_url()` should serve as base for the return values on `get_cwd()` and `get_name()`: In general it should hold:"

   ```
   get url() = get cwd() + / + get name()
   ```

   The *leading* path elements, however, can be `'.'` or `'..'`

   ```
   saga::url src_1 ("ftp://localhost/pub/data/test/info.dat");
   saga::url src_2 ("gridftp://localhost/data/test/info.dat");
   saga::url tgt   ("../../test.txt");

   saga::filesystem file f_1 (src_1);
   saga::filesystem file f_2 (src_2);

   f_1.move (tgt); // tgt is expanded relative to u0
   f_2.move (tgt); // tgt is expanded relative to u1
   ```

8. The exception ordering is now relaxed (from MUST to SHOULD), and a changed order is recommended.

   In particular, the algorithm to find the most specific exception has been changed to ignore the `NotImplemented` exception as long as there were other exceptions thrown. `NotImplemented` will be reported only if there are *only* `NotImplemented` exceptions in the exception list.

   For a detailed discussion on the changed exception ordering, see subsection A.1.

9. The `task.get_result()` semantics has been changed, to act as a universal synchronization point. This makes SAGA task objects more similar to futures. `get_result()` is now waiting, and returning retvals. It also re-throws if the task arrived in a `Failed` state.

10. As in Java's URL class, `url.get_port()` now returns `-1` by default, i.e. if port is unknown.

    ```
    saga::url u ("../tmp/file.txt");
    int port = u.get_port (); // returns -1
    ```

11. The default flag for `open_dir()` is now `Read`.

12. The `task.run()` postcondition is now `'left New state'` instead of `'is in Running state'`, to avoid races with tasks and jobs entering a final state immediately.

13. The job description now supports more JSDL attributes. In particular, `JobProject` and `WallTimeLimit` have been added.

14. The `context.set_default()` method is now gone – its semantics has been added to the `session.add_context()` call, which performs a deep copy of the context first.

    This change breaks backward compatibility. It is implemented in all SAGA implementations.

15. `Read` and `Write` flags are now added to the `saga::namespace` package.

16. The specification now clarifies the URL character escaping mechanisms. A method to return the unescaped string has been added.

17. `close()` is not throwing `IncorrectState` anymore.

18. The specification now clarifies that `object.clone()` does not copy the object id, but assigns a new, unique one.

19. The specification now clarifies, for `namespace::dir.copy (src, tgt)`, that (i) if `src` can be parsed as URL, but contains an invalid entry name, a `BadParameter` exception is thrown, and (ii) if `src` is a valid entry name but the entry does not exist, a `DoesNotExist` exception is thrown.

20. The prototype parameter names for namespace methods have been fixed.

21. The `rpc` constructor argument 'url' now defaults to an empty URL. Also, the parameter name has been fixed.

22. The description of `task_container.cancel (float timeout)` now mentions a default value for timeout (0.0).

23. The SAGA class diagram is now in sync with the class hierarchy defined by the specification.

24. The default value for the `rpc::parameter` size is now fixed (was specified inconsistently).

25. The object type enum does not contain `Exception` anymore, as `exception` does not inherit from `saga::object`.

26. The `saga::job` class now has an attribute which reports its service manager url, as that URL is needed if a `job_service` instance is to be created

in the same security domain, for example. Otherwise, a `job_service` created with no URL can never be reconnected to, e.g. to find previously run jobs:

```
std::string id;
{
  saga::job::service js ();
  saga::job j = js.run_job ("/bin/sleep 1000");
  id = j.get_job_id ();
}
{
  saga:url u = j.get_attribute (''ServiceURL'');
  saga::job::service js (u);
  saga::job j = js.get_job (id);
  j.cancel ();
}
```

27. The `url::translate()` call now accepts an additional session parameter to (a) allow to select eligible backends, and (b) provide security for backend communication.

28. `session.list_contexts()` now returns deep copies of session contexts, not shallow copies. That avoids the change of contexts which are in use.

29. A job executable can now also be searched for in the PATH environment variable, if available.

30. A working directory from a job's `job_description` now gets created if it does not yet exist.

31. several saga classes are now always deeply copied, never shallow. These classes are: `url`, `exception`, `context`, `metric`, `job_description`, and `task_container`.

32. The `stream::server` now has a `connect()` method, which creates a client `stream` instance if the `stream::server` represents a remote connection point.

    This change breaks backward compatibility. It is implemented in all SAGA implementations.

33. The `connect()` calls of both the `stream::service` and `stream::stream` class now support a timeout parameter.

34. It is now clarified that `dir.get_num_entries()` and `dir.list()` are not to include `'.'` and `'..'` .

35. GFD.90 defines several functional packages which all have their own SIDL namespace, amongst them '`file`' and a '`logical_file`'. Those package namespaces are reflected in the implementation name spaces. The two

names above however turned out to be ill chosen: amongst other points, '`file`' clashes with a reserved word in Python, and '`logical_file`' leads to cumbersome class names like '`saga::logical_file::logical_file`', or to confusing names like '`saga::logical_file::logical_directory`'.

Those packages are thus renamed to '`filesystem`' and '`replica`', respectively.

This change breaks backward compatibility. It is implemented in all SAGA implementations.

36. It was clarified what happens to the state of a moved `saga::file` instance.

A.1 Detailed discussion of amended exception ordering

The exception reporting and ordering problem exposed during the implementation of GFD.90 is the single most invasive amendment to the SAGA API Specification. We thus represent here a detailed description and analysis of the problem, as performed by the SAGA group within OGF.

Problem Description:

The exception precedence list in the original specification did not always make sense:

- The NotImplemented exception is actually the least informative one, and should be at the *end* of the list.
- For late-binding implementations, and for implementations with multiple backends in general, it is very difficult to determine generically which exception is more interesting to the end user.

Problem Example

Assume an implementation of the SAGA file API binds (late) to HTTP and FTP.

Assume the following setup: on host `a.b.c`, an http server with `http root` set to `/var/www/`, and an ftp server with its `ftp root` set to `/var/ftp/` are both deployed, using the same credentials for access.

The following files exist, and are owned by root (system permissions in brackets)

```
/var/www/etc/          (x--)
/var/www/etc/passwd    (xxx)
/var/www/usr/          (xxx)

/var/ftp/etc/          (xxx)
/var/ftp/usr/          (x--)
/var/ftp/usr/passwd    (xxx)
```

Assume a SAGA application wants to open `any://a.b.c/etc/passwd`[8] for reading. The WWW backend will throw `PermissionDenied`, the FTP backend will throw `DoesNotExist`.

Both exceptions are correct. There are valid use cases for either exception to be the 'more specific', and thus, in the specifications argumentation, the more

[8] The `any` schema is, in SAGA, a placeholder, which allows the SAGA implementation to autonomously chose a backend and/or protocol to perform the requested operation.

dominant one.

Further, upon accessing `any://a.b.c/usr/passwd`, the situation is exactly inversed. Of course, the implementation will have no means to deduce the intention of the application, and to decide that suddenly the exception from the other backend is more useful.

Problem Diagnosis

The root of the problem is the ability of SAGA to be implemented with late binding. Any binding to a single middleware will result in exactly one error condition, which is to be forwarded to the application. Also, implementations with early bindings can (and indeed will) focus on exceptions which originate from the bound middleware binding for that specific object, and will again be able to report exactly one error condition. (Note that for early binding implementations, the initial operation which causes the implementation to bind to one specific middleware is prone to the same exception ordering problem.)

So it is mostly for late binding implementations that this issue arises, when several backends report errors concurrently, but the standard error reporting mechanism in most languages defaults to report exactly one error condition.

Possible Solutions

A global, predefined ordering of exception will be impossible, or at least arbitrary. The native error reporting facilities of most languages will by definition be inadequate to report the full error information of late binding SAGA implementations.

That leaves SAGA language bindings with three possibilities:

(A) introduce potentially non-native error reporting mechanisms;

Code Example

```
1  saga::filesystem::file f ("any://a.b.c/etc/passwd");
2  std::list <saga::exception> el = f.get_exceptions ();
3
4  // handle all backend exceptions
5  for ( int i = 0; i < el.size (); i++ )
6  {
7    try
8    {
9      throw el[i];
10   }
11   catch ( saga::exception::DoesNotExist )
12   {
```

```
13      // handle exception from ftp backend
14    }
15    catch ( saga::exception::PermissionDenied )
16    {
17      // handle exception from www backend
18    }
19  }
```

(B) acknowledge the described limitation, document it, and stick to the native error reporting mechanism;

Code Example

```
1   try
2   {
3     saga::filesystem::file f ("any://a.b.c/etc/passwd");
4   }
5   catch ( saga::exception::DoesNotExist )
6   {
7     // handle exception from ftp backend
8   }
9   catch ( saga::exception::PermissionDenied )
10  {
11    // handle exception from www backend (which will not be forwarded in
12    // our example, this this will never be called)
13  }
```

(C) a mixture of (A) and (B), with (B) as default.

Code Example

```
1   try
2   {
3    saga::filesystem::file f ("any://a.b.c/etc/passwd");
4   }
5   catch ( saga::exception::DoesNotExist e )
6   {
7     cout << e.what ();
8     cout << e.get_message  ();
9     // DoesNotExist     ...
10
11    std::list <std::string> ml = e.get_all_messages ();
12    // DoesNotExist     ...
13    // PermissionDenied ...
14    // NotImplemented   ...
15
```

```
16    return;
17  }
18  catch ( saga::exception e)
19  {
20    // handle all backend exceptions
21    std::list <saga::exception> el = e.get_all_exceptions ();
22    // DoesNotExist  // no infinite recursion
23    // PermissionDenied
24    // NotImplemented
25
26    for ( int i = 0; i < el.size (); i++ )
27    {
28      try
29      {
30        throw el[i];
31      }
32      catch ( saga::exception::DoesNotExist )
33      {
34        // handle exception from ftp backend
35        std::list <saga::exception> el = e.get_all_exceptions ();
36        // exception from backend A
37        // exception from backend B
38        // ...
39
40        std::list <saga::string> el = e.get_all_messages ();
41        // exception message from backend A
42        // exception message from backend B
43        // ...
44      }
45      catch ( saga::exception::PermissionDenied )
46      {
47        // handle exception from www backend
48      }
49      // ...
50    }
51  }
```

Note that **(C)** may not be possible in all languages.

Discussion of the C++ Bindings

C++ is actually be able to implement **(C)**. The C++ bindings would then introduce a `saga::exception` class, and the respective sub classes, which represent the 'most informative/specific' exception. How exactly the 'most informative/specific' exception is selected from multiple concurrent implementations is left to the implementation, and cannot sensibly be prescribed by the specification nor the language binding, as discussed above. (The spec could propose such

a selection algorithm though). However, the `saga::exception` class would have the additional ability to expose the full set of backend exceptions, for example as list:

```
std::list <saga::exception> exception.get_all_exceptions ();
```

Further, it would be advisable (for all language bindings actually) to include *all* error messages (from all backend exceptions) into the error message of the top level exception (this is already implemented in CCT's C++ implementation):

```
catch ( saga::exception e )
{
  std::cerr << e.what ();
}
```

would print the following message:

```
exception (top level): DoesNotExist
  exception (ftp): DoesNotExist - /etc/passwd does not exist
  exception (www): PermissionDenied - access to /etc denied
```

Conclusion

The group decided to move for option **(C)**, and the sPecification was amended accordingly.

References

[1] SAGA Web Pages.

[2] BOOST C++ Libraries. `http://www.boost.org/`.

[3] C. Catlett. GFD.1 – Global Grid Forum Documents and Recommendations: Process and Requirements. GGF Community Praxis, Global Grid Forum, 2001.

[4] T. Cortes, C. Franke, Y. Jégou, T. Kielmann, D. Laforenza, B. Matthews, C. Morin, L. Prieto, A. Reinefeld, D. Level, et al. XtreemOS: a Vision for a Grid Operating System. White paper, 2008.

[5] T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, A. Merzky, J. Shalf, and C. Smith. GFD.90 – SAGA Core API Specification. OGF Proposed Recommendation, Open Grid Forum, 2007.

[6] S. Hirmer, H. Kaiser, A. Merzky, A. Hutanu, and G. Allen. Generic support for bulk operations in grid applications. In Proceedings of the 4th international workshop on Middleware for grid computing, page 9. ACM, 2006.

[7] S. Jha, H. Kaiser, Y. El Khamra, and O. Weidner. Design and Implementation of Network Performance Aware Applications using Saga and Cactus. In 3rd IEEE Conference on eScience2007 and Grid Computing, Bangalore, India. Citeseer, 2007.

[8] S. Jha, H. Kaiser, A. Merzky, and O. Weidner. Grid Interoperability at the Application Level using Saga. In IEEE International Conference on e-Science and Grid Computing, pages 584–591, 2007.

[9] H. Kaiser, A. Merzky, S. Hirmer, and G. Allen. The SAGA C++ Reference Implementation. In Second International Workshop on Library-Centric Software Design (LCSD'06), page 101. Citeseer, 2006.

[10] H. Kaiser, A. Merzky, S. Hirmer, G. Allen, and E. Seidel. The SAGA C++ Reference Implementation: a Milestone toward new High-Level Grid Applications. In Proceedings of the 2006 ACM/IEEE conference on Supercomputing, page 184. ACM, 2006.

[11] A. Luckow, L. Lacinski, and S. Jha. SAGA BigJob: An Extensible and Interoperable Pilot-Job Abstraction for Distributed Applications and Systems.

[12] A. Merzky, K. Stamou, and S. Jha123. Application Level Interoperability between Clouds and Grids. In Proceedings of the 2009 Workshops at the Grid and Pervasive Computing Conference, pages 143–150. IEEE Computer Society, 2009.

[13] R. Sirvent, A. Merzky, R. Badia, and T. Kielmann. GRID SuperScalar and SAGA: Forming a High-Level and Platform-Independent Grid Programming Environment. In CoreGRID Integration WorkShop, volume 2005. Citeseer, 2005.

[14] R. van Nieuwpoort, T. Kielmann, and H. Bal. User-Friendly and Reliable Grid Computing Based on Imperfect Middleware. In Proceedings of the ACM/IEEE Conference on Supercomputing (SC07), 2007.