The SAGA C++ Reference Implementation

Lessons Learnt from Juggling with Seemingly Contradictory Goals

Hartmut Kaiser
Louisiana State University
Baton Rouge
Louisiana, USA
hkaiser@cct.lsu.edu

Andre Merzky
Vrije Universiteit
Amsterdam
Netherlands
andre@merzky.net

Stephan Hirmer
Louisiana State University
Baton Rouge
Louisiana, USA
shirmer@cct.lsu.edu

Gabrielle Allen
Louisiana State University
Baton Rouge
Louisiana, USA
gallen@cct.lsu.edu

ABSTRACT

The Simple API for Grid Applications (SAGA) is an API standardization effort within the Open Grid Forum (OGF). OGF strives to standardize grid middleware and grid architectures. Many OGF specifications are still in flux, and multiple, incompatible grid middleware systems are used in research or production environments. SAGA provides a simple API to programmers of scientific applications, with high level grid computing paradigms which shield from the diversity and dynamic nature of grid environments.

The SAGA specification scope will be extended in the coming years, in sync with maturing service specifications. SAGA is defined in SIDL (Scientific IDL). A C++ language binding is under development, language bindings for FORTRAN, Java, Python and C are planned.

Implementing the SAGA API specification is an interesting and challenging problem itself, due to the dynamic environment presented by current grids. Nevertheless, the perceived need of the grid community for a high level API is great enough to tackle that problem *now*, and not to wait until the standardization landscape settles. This paper describes how the C++ SAGA reference implementation tries to cope with these requirements – we believe there are lessons to learn for other API implementations.

1. INTRODUCTION

Relatively few existing grid-enabled applications exploit the full potential of grid environments. This is mainly caused by the difficulties faced by programmers trying to master the complexities of grids (see section 2). Several projects concentrate on the development of high-level, application-oriented toolkits that free programmers from the burden of adjusting their software to different and changing grids. The Simple API for Grid Applications (SAGA) [1] is a prominent recent API standardization effort which intends to simplify the development of grid-enabled applications, even for scientists with no background in computer science, or grid computing. SAGA was heavily influenced by the work undertaken in the GridLab project [2], in particular by the Grid Application Toolkit (GAT) [3], and by the Globus Commodity Grid [4]. The concept of high level grid APIs has proved to be very useful in several projects developing cyberinfrastructures, such as the SURA Coastal Ocean Observing Program (SCOOP) which uses GAT to interface to large data archives [5] using multiple access protocols.

The C++ implementation of the SAGA API presented in this paper leverages the experience we obtained from developing the GAT and will provide a reference implementation for the OGF standardization process. As the SAGA API is originally specified using the Scientific Interface Description Language (SIDL) [6], the implementation also represents a first attempt to develop the SAGA C++ language bindings. It has a number of key features, described later in detail:

- Synchronous, asynchronous and task oriented versions of every operation are transparently provided.
- Dynamically loaded adaptors bind the API to the appropriate grid middleware environment, at runtime. Static pre-binding at link time is also supported.
- Adaptors are selected on a call-by-call basis (late binding, supported by a object state repository), which allows for incomplete adaptors and inherent fail safety.
- Latency hiding (e.g. asynchronous operations and bulk optimizations) is generically and transparently applied.
- A modular API architecture minimizes the runtime memory footprint.
- API extensions are greatly simplified by a generic call routing mechanism, and by macros resembling (SIDL) [6] used in the SAGA specification.
- Strict adherence to Standard-C++ and the utilization of Boost [7] allows for excellent portability and platform independence.

2. REQUIREMENTS

As mentioned in the introduction, the SAGA C++ reference implementation must cope with a number of very dynamic requirements. Additionally, it must provide the "simple" and "easy-to-use" API the SAGA standard is intended to specify. We describe the resulting requirements in some detail motivating our SAGA implementation design described in section 3.

We identified several main characteristics the SAGA C++ reference implementation must provide, if any of these properties is missing, acceptance in the targeted user community will be severely limited:

- It must cope with evolving grid standards and changing grid environments.

- It must be able to cope with future SAGA extensions, without breaking backward compatibility.
- It must shield application programmers from the evolving middleware, and X should allow various incarnations of grid middleware to co-exist.
- It must actively support fail safety mechanisms, and hide the dynamic nature of resource availability.
- It must be portable and, both syntactically and semantically, platform independent.
- It must allow these and other latency hiding techniques to be implemented.
- It must meet other end user requirements outside of the actual API scope, such as ease of deployment, ease of configuration, documentation, and support of multiple language bindings.

3. GENERAL DESIGN

The implementation level requirements of the SAGA reference implementation as described in the previous section directly motivate a number of design objectives. Our most important objective was to design a state-of-the-art Grid application framework satisfying the majority of user-needs while remaining as flexible as possible.

This flexibility and extensibility of the implementation, is then central to the design, and dominates the overall architecture of the library (see figure 1). As a summary: only components known to be stable, such as the SAGA "look & feel" and the SAGA utility classes, are statically included in the library – all other aspects of the API implementation, such as the core SAGA classes and the middleware compile time and run time bindings, are designed to be components which can be added and selected separately.

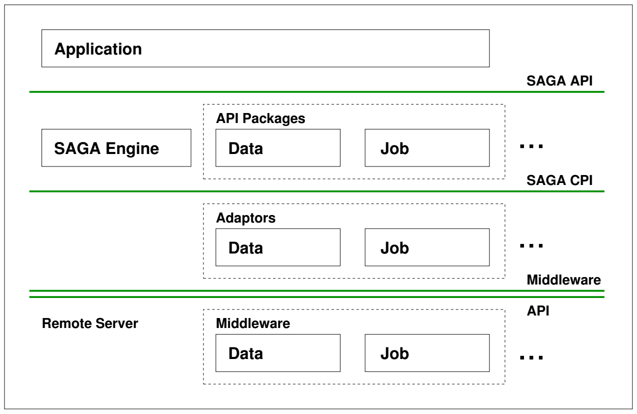

Figure 1: Architecture: A lightweight engine dispatches SAGA calls to dynamically loaded middleware adaptors. See text for details.

3.1 Design Objectives

Although the Simple API for Grid Applications is, by definition *simple* for application developers, this doesn't imply that the implementation itself has to be simple. We made a major effort to build as much logic and functionality as possible into the SAGA library core, providing all the needed common functionality. This enables the user to extend it with minimal effort. On the other hand, the library *is* designed to be easy to build, use, and deploy.

A major design objective was to maximize decoupling of different components of the developed library to provide as much *flexibility*, *adaptability* and *modularity* as possible.

As the SAGA implementation is expected to be used on different platforms and operating systems we strive for maximal implementation *portability.*

The API should be *extensible* with minimal effort: ideally, adding a new API class is orthogonal to all other properties of the implementation.

3.2 The Overall Architecture

To meet these goals we decided to decouple the library components in three completely orthogonal dimensions – the user of the library may use and combine these freely and develop additional suitable components usable in tight integration with the provided modules.

3.2.1 Horizontal Extensibility – API Packages

Our implementation uses the grouping of sets of API functions as defined by the SAGA specification to define *API packages.* Current packages are: file management, job management, remote procedure calls, replica management, and data streaming. These modules depend only on the SAGA engine, the user is free to use and link only those actually needed by the application, minimizing the memory footprint. It is straightforward to add new packages (as the SAGA specification is expected to evolve) since all common operations needed inside these packages (such as adaptor loading and selection, or method call routing) are imported from the SAGA engine. The creation of new packages is essentially reduced to: (1) add the API package files, and declare the classes, (2) reflect the SAGA object hierarchy (see section 4.1.2), and (3) add class methods.

The declaration and implementation of the API methods is simplified by macros, which essentially correspond directly to the methods SIDL specification (see section 4.6). We are considering (partly) automating new package generation, by parsing the SIDL specification and generating the class stubs and class method specifications. Additionally, this approach will also allow us to generate other SAGA language bindings from the SIDL specification, such as for C and FORTRAN.

3.2.2 Vertical Extensibility – Middleware Bindings

A layered architecture (see figure 1) allows us to vertically decouple the SAGA API from the used middleware. Separate adaptors, either loaded at runtime, or pre-bound at link time, dispatch the various API function calls to the appropriate middleware. These adaptors implement a well defined *Capability Provider Interface* (CPI) and expose that to the top layer of the library, making it possible to switch adaptors at runtime, and hence to switch between different (even concurrent) middleware services providing the requested functionality.

The top library layer dispatches the API function calls to the corresponding CPI function. It additionally contains the *SAGA engine* module, which implements: (1) the core SAGA objects such as session, context, task or task_container – these objects are responsible for the SAGA look & feel, and are needed by all API packages, and (2) the common functions to load and select matching adaptors, to perform generic call routing from API functions to the selected adaptor, to provide necessary fall back implementations for the synchronous and asynchronous variants of the API functions (if these are not supported by the selected adaptor).

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

3.2.3 *Extensibility for Optimization and Features*

Many features of the engine module are implemented by intercepting, analyzing, managing, and rerouting function calls between the API packages, (where they are issued) and the adaptors (where they are executed and forwarded to the middleware). To generalize this management layer, a PIMPL [8] (Private IMPLementation) idiom was chosen, and is rigorously used throughout the SAGA implementation. This PIMPL layering allows for a number of additional properties to be transparently implemented, and experimented with, without any change in the API packages or adaptor layers. These features include: generic call routing, task monitoring and optimization, security management, late binding, fallback on adaptor invocation errors, and latency hiding mechanisms. The decoupling of these features from the API and the adaptors succeeds, essentially, because these properties affect only the IMPL side of the PIMPL layers.

The engine module is thus fully generic, and loosely coupled to both the API and adaptor layers. Any engine feature, all optimizations, latency hiding techniques, monitoring features etc. are implemented in generically, and are orthogonal to the API and adaptor extensions.

4. IMPLEMENTATION DETAILS

The following section will describe certain implementation details of the SAGA C++ reference implementation. As will be described, the implementation gains its flexibility mainly from the combined application of C++'s compile time and runtime polymorphism features, i.e. template's and virtual functions respective.

4.1 General Considerations

To achieve maximum portability, platform independence and code reuse, the SAGA C++ reference implementation relies strictly on the Standard C++ language features, and uses the C++ Standard and Boost libraries where possible.

4.1.1 *The SAGA task model*

A central concept of the SAGA API design is the SAGA task model [9], prescribing the form of synchronous and asynchronous method calls. Essentially, each method call comes in three variants: as a *synchronous call* (executed immediately), as a *asynchronous call*, and as a *task call*. The latter versions of the calls return a `saga::task` class instance. A `saga::task` thus represents an asynchronously running operation, and has an associated state (`New, Running, Finished, Failed`). Task versions of the method calls return a `New` task, asynchronous versions return a `Running` task. For symmetry reason, we added a fourth, synchronous version of method calls, returning a `Finished` task. The realization of the `saga::impl::task` class bases on a implementation of the *futures* paradigm, a concurrency abstraction first proposed for MultiLisp [10]. The C++ rendering of the SAGA task model is shown in figure 2.

```
SAGA task model

string dest = "any://host.net//data/dest.dat";
file   file  ("any://host.net//data/src.dat");

// normal sync version of the copy method
file.copy (dest);

// the three task versions of the same method
task t1 = file.copy <task::Sync>  (dest);
task t2 = file.copy <task::ASync> (dest);
task t3 = file.copy <task::Task>  (dest);

// task states of the returned saga::task
// t1 is in 'Finished' or 'Failed' state
// t2 is in 'Running'              state
// t3 is in 'New'                  state

t3.run  ();
t2.wait ();
t3.wait ();
// all tasks are 'Finished' or 'Failed' now
```

Figure 2: The SAGA task model rendered in C++

While we tried to absolutely minimize the use of template's in the API layer, it was decided to implement the different flavors of the API functions using function templates (see figure 2). This makes the whole SAGA C++ implementation *generic* with respect to the synchronicity model, being another reason for providing two types of the synchronous function flavors: a direct and a task based one.

4.1.2 *The Object Instance Structure*

As already mentioned, the SAGA API objects are implemented using the PIMPL idiom. Their only essential member is a `boost::smart_ptr<>` to the base class of the implementation object instance[1], keeping it alive. This makes them very lightweight and copyable without major overhead, and therefore storable in any type of container.

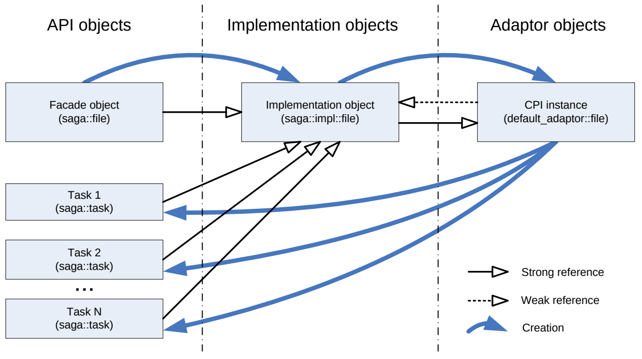

Figure 3: Object instance structure: Copying a API object instance means sharing state, returned tasks keep implementation alive.

As shown in figure 3, any API object instance creates the corresponding impl instance holding all the instance data of the SAGA object instance Copying of an API instance therefore shares this state between the copied instances. This behavior is consistent with anticipated handle based SAGA language bindings (e.g. in C or FORTRAN), where copying the handle representing a SAGA object instance naturally

[1]We refer to the implementation side of the PIMPL layer as *impl classes* in this document

means sharing the internal instance data as well[2].

Due to the shared referencing after copies, the impl instances can be kept alive by objects which depend on their state – for example, a task keeps the objects alive for which they represent a asynchronous method call (see figure 3).

The call sequence for creating a SAGA API object instance is shown in figure 4. Whenever needed, the implementation creates a CPI object instance implemented in one of the adaptors. The process of adaptor selection and CPI instantiation is injected into the API packages by the macros mentioned before (see section 3.2.1).

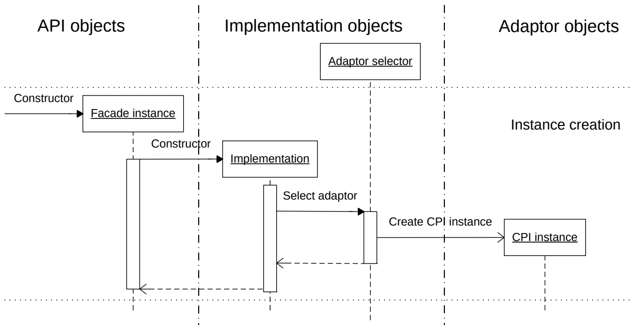

Figure 4: Object creation: Sequence diagram depicting the creation of all components as showed in figure 3. Note, how the call is intercepted by a SAGA engine module component to select a appropriate adaptor.

4.2 Inheritance and PIMPL

An interesting problem in the strict application of the PIMPL mechanism lies in the API object hierarchy: the `saga::file` class for example inherits the `saga::ns_entry` class, which inherits the `saga::object` class. Additionally, the SAGA specification requires all these classes to implement additional interfaces. Now, the PIMPL paradigm requires all class instances to own exactly *one* impl pointer[3], and are built using single inheritance only, otherwise we would face object slicing problems when copying around the base classes only. The solution is (1) to add the required interfaces to the most derived classes by duplication the interface functions, and (2) to up-cast the impl reference stored in the base class whenever needed.

API classes access the impl pointer through `get_impl()`, which, in derived classes, implies a static up-cast for the base class' impl pointer.

The implementation objects resemble the API object hierarchy. These are also derived from a common base class and contain, somewhere in their own hierarchy, similar objects to the API objects. The `saga::impl::file` class[4] inherits the `saga::impl::ns_entry` class, which inherits the implementation specific `saga::impl::proxy` class, which is

[2]A polymorphic `saga::object::clone()` method is, however, part of the SAGA API, and allows for explicit deep copies of API objects, forcing the instance data to be copied as well.

[3]In fact the impl pointer stored in any `saga::object` instance is a `boost::smart_ptr<saga::impl::object>`, i.e. a reference to the very base class of the implementation object hierarchy.

[4]The `saga::impl::file` class for example is the implementation equivalent to the `saga::file` class, as we kept all API classes in `namespace saga` and all corresponding implementation classes in `namespace saga::impl`.

Constructors in the saga::file hierarchy

```
// saga::file constructor
file::file ([args])
:  ns_entry (new saga::impl::file ([args]))  {}

// saga::ns_entry constructor
ns_entry::ns_entry (saga::impl::ns_entry* impl)
:  saga::object (impl) {}

// saga::object constructor
//  'impl_' is a boost::smart_ptr<saga::impl::object>
object::object (saga::impl::object* impl)
:  impl_ (impl) {}
```

Figure 5: Realizing inheritance in PIMPL classes (simplified). Only the `saga::object` base class owns an impl pointer.

derived from the common `saga::impl::object` class. Thus, the class hierarchy on the implementation side of the PIMPL paradigm reflects the API side of the class hierarchy, ensuring the correct casting behavior in the `get_impl()` methods.

4.3 State Management

Section 4.1.2 discussed object state, in relation to state sharing of objects after shallow copies. Here we describe the object state management of the SAGA implementation in more detail, since state management is a central element on several layers. On a different layer, the adaptors represent operations on the object instances, and need to maintain state as well. At the adaptor level this is complicated by the fact that the object state can (and in general will) be changed by several adaptors (remember: adaptors are selected at runtime, and may change for each API function invocation). For state management, we hence distinguish between three types of state information.

- *Instance data* represent the state of API objects (e.g. file name, file pointer etc.). These are predefined and not amendable by the adaptor as they represent common data either passed from the constructor, or needed for consistent state management on the API level.
- *Adaptor data* represent the state of CPI objects (e.g. open connections) and are shared between all instances of all CPI object types implemented by a single adaptor and corresponding to a single adaptor instance.
- *Adaptor-instance data* represent the state shared between all CPI instances created for a single API object and implemented by the same adaptor (e.g. remote handles).

The lifetime of any type of the state information is maintained by the SAGA engine module, which significantly simplifies the writing of adaptors.

All three types of state information are carefully protected from race conditions potentially caused by the multithreaded nature of the implementation. We provide helper classes simplifying the correct locking of the instance data. This uniform state management enables object state persistency in the future, with minimal impact on the code base.

4.4 Generic Call Routing

The essential idea of the implemented generic API call routing mechanism is to represent the calls as abstract objects, and to redirect their execution depending on several attributes and the adaptor suitability. For example, an asynchronous method call for a `saga::file` instance is preferably directed to a asynchronous file adaptor, or, if such is not available, to a synchronous file adaptor, wrapping it in a thread, or, returns an error otherwise (`NotImplemented`).

This routing mechanism allows for (1) trivial (synchronous) adaptor implementations, (2) late binding (differents adaptor can be selected for each call, even on the same API object instance), (3) variable adaptor selection strategies (based on adaptor meta data, user preferences, and heuristics), and (4) latency hiding (bulk optimization [11], or automatic load distribution over multiple adaptors). Figure 6 is depicting the injection of the call routing mechanism by the SAGA engine.

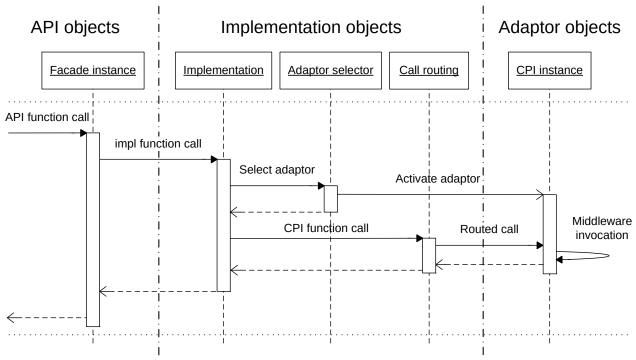

Figure 6: API function call: Diagram illustrating the execution sequence through the different object instances during a call to any adaptor supplied function.

All SAGA API methods come in synchronous and asynchronous flavors (see section 4.1.1). To avoid, that adaptors need to implement both flavors, we provide fallback implementations in the SAGA engine. The synchronous behavior is modelled by executing the the asynchronous implementation and waiting for it to finish. The asynchronous wraps the synchronous implementation into a thread representing the asynchronous remote operation.

Even if this approach has a couple of drawbacks (it is not really asynchronous, the middleware call still blocks, causing lock problems if implemented badly, and tasks are not able to survive the application life time), the mechanism simplifies adaptor implementations greatly, as most of the existing grid middleware is *not* fully asynchronous anyway.

4.5 Adaptor Selection

The selection of suitable adaptors at runtime is a central component in the implementation (see figure 6). It is, a very simple mechanism: on loading, the adaptor components register their *capabilities* in the adaptor registry. If a method is to be executed, the adaptor selector searches that registry for all suitable adaptors, orders them, and tries them one-by-one, until the method invocation succeeds. The adaptor selection is routed through SAGA engine, generically implementing this for any API function.

To overcome the limitations of this approach (several CPI instances have to be created, remote operations add additional latencies), our library allows adaptors to specify additional, key/value based meta data, and also allows to exchange the adaptor selection component.

4.6 Utilization of Macros

Our SAGA implementation makes extensive use of C++ preprocessor macros. This might be perceived as a design flaw, at least by some readers, and we were very hesitant to utilize macros extensively. However, the benefits for the end user and other programmers(!) seem currently to outweigh the problems, such as limited debugging abilities. mentioned in section 3.2.1, We are using Boost.Wave [7] features to pregenerate partially macro expanded sources to overcome the disadvantages of plain macros, hence simplifying debugging and improving readability.

5. IMPLEMENTATION PROPERTIES

This section summarizes the properties of our SAGA implementation from an end user perspective, gives an overview about the lessons learnt, and motivates further developments and extensions.

5.1 Uniformity over Programming Languages

The SAGA API specification is language independent. One of the consequences of this is that it does not use templates, which were thought too difficult to express uniformly over many languages. Also, the specification tries to be concise about object state management, and hence also expresses semantics for shallow and deep copies. Our implementation follows the SAGA API specification closely. It is also designed to accommodate wrappers in other languages. A Python wrapper for our library is in alpha status, and we plan to add wrappers to provide bindings to C, FORTRAN, Perl, and possibly others. In the past we found it very useful to be able to write Python adaptors for the GAT [3], a predecessor of SAGA, and we will provide similar support here as well.

5.2 Genericity in Respect to Middleware, and Adaptability to Dynamic Environments

The dynamic nature of grid middleware is addressed in our implementation by the described adaptor mechanism which binds to diverse middleware. Late binding, fall back mechanisms, and flexible adaptor selection allow for additional resilience against an dynamic and evolving run time environment. Adaptors need to deploy mechanisms like resource discovery, and need to implement fully asynchronous operations, if the complete software stack is to be able to cope with dynamic grids.

5.3 Modularity ensures Extensibility

Section 4.6 described how the SAGA implementation will be able to cope with the expected evolution and extension of the SAGA API. The adaptor mechanism allows for easy extensions of the library, providing additional middleware bindings. The task of adaptor writing requires massively more effort than the implementation of the presented library. Ideally, middleware vendors will *implement* adaptors for SAGA, and deliver them as part of their client side software stack. This would be a major step towards wide spread grid applications.

5.4 Portability and Scalability

Heterogeneous distributed systems naturally require portable code bases. Our library implementation is very portable, as we strictly adhere to the C++ standard and portable libraries. We currently develop the library on Windows, Linux and MacOS concurrently, covering three major target platforms without any problems. However, the portability of our SAGA implementation depends on the portability of the adaptors, and hence on the portability of the grid middleware client interfaces, being the much greater problem if compared to the library code itself.

Distributed applications are often sensitive to scalability issues, in particular in respect to remote communications. This equally applies to SAGA, so that scalability concerns are naturally raised in respect to SAGA implementations as well. Even if the SAGA API is not targeting high performance communication schemes, but tries to stick to simple communication paradigms, our design allows for zero-copy implementations of the SAGA communication APIs, and for fast asynchronous notification on events – both are deemed critical for implementing scalable distributed applications.

5.5 Simplicity for the End User

SAGA is *designed* to be simple. However, simplicity of an API is not only determined by its API specification, but also by its implementation: simple deployment and configuration, resilience against lower level failures, adaptability to diverse environments, stability, correctness, and peaceful coexistence with other programming paradigms, tools and libraries are some of the characteristics which need attention while implementing the SAGA API.

A modular implementation helps to keep a library implementation itself simple. Features as the generic call routing, or the adaptor selection are hidden in the engine module. Modeling these central properties as modules increases the readability and maintainability of the code significantly.

Due to its notion of tasks the SAGA API implicitly introduces a concurrent programming model. Our C++ language binding of the API, allows to combine that model with arbitrary mechanisms for managing concurrent program elements (thread safety, object state consistency, etc.).

6. FUTURE WORK

As mentioned, work on appropriate middleware adaptors will undoubtedly require significant resources in the future. This motivates us to work on simplifying adaptor creation, integration and maintenance, and seek support and contributions from the OpenSource community, and from grid middleware vendors. We will develop other language bindings on API and adaptor level, and apply further generic latency hiding techniques.

7. CONCLUSION

We have described the C++ reference implementation of the SAGA API, which is designed as a generic and extensible API framework: it allows for the extension of the SAGA API, easily usable for other APIs); it allows for run-time extension of middleware bindings, and it allows for orthogonal optimizations and features, such as late binding, diverse adaptor selection strategies, and latency hiding. The used techniques enable these features, amongst them the application of the PIMPL paradigm for a complete class hierarchy and generic call routing.

These implementation techniques incur a certain overhead, however, in grid environments the runtime overhead is usually vastly dominated by communication latencies, so that ***this** overhead does not matter.* The lesson learned is that distributed environments *allow* for fancy mechanisms, which are too expensive in local environments. Fail safety and latency hiding mechanisms are more important than, for example, virtual functions, late binding, and additional abstraction layers.

8. ACKNOWLEDGMENTS

We thank Thorsten Schütt for his work on Futures; Michel Zsandstra and Vladimir Prus for their initial work on dynamic loading; the SAGA groups in OGF for their work on the SAGA specification; the Center for Computation & Technology at Louisiana State University, the Vrije Universiteit Amsterdam, and the XtreemOS project for funding this work.

9. REFERENCES

[1] SAGA Core Working Group. Simple API for Grid Applications – API Version 1.0. Technical report, OGF, 2006. `http://forge.ggf.org/sf/projects/saga-core-wg`.

[2] Gridlab: A Grid Application Toolkit and Tsetbed. `http://www.gridlab.org/`.

[3] G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A. Merzky, R. Nieuwpoort, A. Reinefeld, F. Schintke, T. Schütt, E. Seidel, and B. Ullmer. The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid. *Proceedings of the IEEE*, 2004.

[4] G. v. Laszewski, I. Foster, J. Gawor, and P. Lane. A Java commodity grid kit. *Concurrency and Computation: Practice and Experience*, 13(8–9):645–662, 2001.

[5] D. Huang and G. Allen and C. Dekate and H. Kaiser and Z. Lei and J. MacLaren. getdata: A Grid Enabled Data Client for Coastal Modeling. In *High Performance Computing Symposium (HPC 2006)*, April 3–6 2006.

[6] SIDL. Scientific Interface Definition Language. `http://www.llnl.gov/CASC/components/babel.html`.

[7] Boost C++ libraries. `http://www.boost.org/`.

[8] H. Sutter. Pimples–Beauty Marks You Can Depend On. *C++ Report*, 10(5), 1998. `http://www.gotw.ca/publications/mill04.htm`.

[9] T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, G. v. Laszewski, C. Lee, A. Merzky, H. Rajic, and J. Shalf. SAGA: A Simple API for Grid Applications – High-Level Application Programming on the Grid. *Computational Methods in Science and Technology: special issue "Grid Applications: New Challenges for Computational Methods"*, 8(2), SC05, November 2005.

[10] R. H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[11] S. Hirmer, H. Kaiser, A. Merzky, A. Hutanu, and G. Allen. Seamless Integration of Generic Bulk Operations in Grid Applications. In *Submitted to International Workshop on Grid Computing and its Application to Data Analysis (GADA'06)*, Agia Napa, Cyprus, 2006. Springer Verlag.