The SAGA C++ Reference Implementation

Lessons Learnt from Juggling with Seemingly Contradictory Goals

Hartmut Kaiser
Center for Computation & Technology
Louisiana State University
Baton Rouge, Louisiana, USA
hkaiser@cct.lsu.edu

Andre Merzky
Vrije Universiteit, Amsterdam
Amsterdam, The Netherlands
andre@merzky.net

Stephan Hirmer
Center for Computation & Technology
Louisiana State University
Baton Rouge, Louisiana, USA
shirmer@cct.lsu.edu

ABSTRACT

The Simple API for Grid Applications (SAGA) is an ongoing API standardization effort within the Open Grid Forum (OGF). OGF strives to standardize grid middleware, meaning grid service interfaces, grid enabled protocols, and general grid architecture. Many grid standard specifications are still in flux, and there are multiple, incompatible grid middleware systems deployed in research or production environments. SAGA provides a simple API to programmers of scientific applications, allowing them to use high level grid computing paradigms, and providing a shield from the diversity and dynamicity of grid environments.

The SAGA specification should extend in scope over the next couple of years, in sync with maturing service specifications. SAGA is defined in SIDL (a language independent interface description language), a C++ language binding is already being developed, language bindings for FORTRAN, Java, Python and C are planned.

Actually implementing the SAGA API specification is an interesting and challenging problem itself, due to the 'dynamic' (or even chaotic) requirements presented by current grid environments. 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 think there are lessons to learn for other API implementations.

1. INTRODUCTION

With the increasing demand for computational power, the collaborative use of geographically dispersed computing resources (i.e. computational grids) has become increasingly popular. Still, relatively few grid-enabled applications exist that exploit the full potential of grid environments. This is mainly caused by the difficulties faced by programmers trying to get on top of the related complexities (see section 2).

With the right grid resources and middleware in place, research now concentrates 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] – one of the first major attempts to build a high level API to grid services. The concept of the GAT has proved to be very useful in several projects developing cyberinfrastructures such as the SURA Coastal Ocean Observing Program (SCOOP), for instance allowing to build a tool interfacing to a large data archive [4] using multiple access protocols.

The C++ implementation of the SAGA API presented in this paper leverages the experience we got from developing the GAT and will provide a reference implementation for the OGF standardization process. It also represents a first attempt to develop the SAGA C++ language bindings. It has a number of key features, described in more detail later in the text:

- 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), which allows for adaptors with reduced capabilities, and provides inherent fail safety. A generic object state repository supports the late binding.
- Latency hiding schemes such as asynchronous operations and bulk optimizations are generically and transparently applied, even if not explicitly supported by the adaptors or the middleware.
- A modular API architecture minimizes the runtime memory footprint.
- API extensions are greatly simplified by the encapsulation of a generic call routing mechanism, and by macros resembling the Scientific Interface Description Language (SIDL) [5] used in the SAGA specification.
- Strict adherence to Standard-C++ and the utilization of Boost [6] 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.

2.1 Dynamic Specification Landscape

The Open Grid Forum (OGF) [7] is an international standardization body whose primary objective is to define a set of standards in the emerging field of grid computing. OGF specifications will cover grid architectures, protocols, interfaces, and APIs. However, the whole field is young, and the complexity of grids is not yet completely understood, either in terms of academic research or for industrial and commercial applicability and impact. This fact, along with the complexity of the problem itself, causes the grid specification landscape to evolve slowly: it has several significant gaps, and it is widely expected that existing specifications will change [8]. The time needed for grid standards to stabilize is estimated to be 5 to 10 years.

The expectations for grid computing to solve real world problems remains very high, partly due to the initial enthusiasm (or hyping) in the field. This is to the frustration of end users with distributed environments in general (scalability and interoperability is still, after many years, a very difficult problem on many layers). These observations imply the necessity of an interface abstraction for early adopters shielding the implementers of grid applications from the evolving grid standardization landscape, and allows for a migration path to later grid systems with assessable effort.

A SAGA implementation must cope with evolving grid standards and changing grid environments.

2.2 Evolving SAGA Specification

The SAGA specification itself is currently limited and expected to expand in scope over time. In particular in respect to new emerging grid service standards it is expected that new SAGA extensions will be required to provide these programming paradigms to the application developers. The general look & feel of the SAGA specification is, however, thought to be more stable, and there is hope that extensions are merely semantic (new objects, new method calls), but with limited or no syntactical additions (no change to the object model, or the task model etc.).

A SAGA implementation must be able to cope with future SAGA extensions easily, without breaking support and backward compatibility for early SAGA adopters and applications.

2.3 Evolving Grid Middleware

The evolution of grid standards as described in 2.1 implies that implementations of these standards are evolving as well, and very much so. In fact, the major Grid middleware system used over the last 8 years or so, Globus [9], went from version 1.0 to 4.0, thereby undergoing significantly more interoperability breaking updates than the major version numbers suggest. Evolutions of other grid middlewares does not differ in that respect significantly, unless it was developed for very specific environments and purposes.

These software systems are, on the other hand, large projects and well funded, and invest significant effort in training and support. Smaller systems, research developments, and standard reference implementations have, in general, the same problem, but much less resources to limit the impact of that evolution process for the end user. Industrial/commercial implementations with well defined migration paths and the usually accompanying professional support are, in reality, to be counted on the fingers of one hand.

Any high level grid API implementation, such as a SAGA implementation, must shield application programmers from the evolving middleware implementations, and in particular should allow various incarnations of grid middleware to co-exist.

2.4 Dynamic Grid Environment

As grid middleware evolves, deployed grid environments face constant changes of middleware deployments (new versions and services are rolled out frequently, often with unclear migration paths). Grid environments are dynamic by design, with respect to the availability of services and other resources. Any application designed to run on grids should be aware of this property, and implement fail safety mechanisms, and not rely on the static availability of resources. Much of that flexibility however can (and should, in our opinion) be hidden from the application programmer. For example, an upgrade in a services protocol version should be handled in the client libraries talking to the service and not at the application level. Resource discovery, fail safety on service failures and simple fallbacks such as redundant service deployments are other examples of mechanisms vital for grid applications, but should not need explicit reflection in application code.

A SAGA implementation must allow for and, where possible, actively support fail safety mechanisms, and hide the dynamic nature of grid resource availability from the application.

2.5 Heterogeneous Grid Environment

The dynamicity of grid environments is also reflected in their (at least potential) heterogeneous nature: although most deployed grids focus on Linux based clusters, grids are designed to cope with any OS (real or virtual), on any resource. The predominance of Linux is rather a indication of the prematurity of grid middleware developments than an intentional design artifact.

A SAGA implementation must be portable and, both syntactically and semantically, platform independent.

2.6 Distributed Grid Applications

Within the domain of distributed applications, which always imply remote communication, latencies considerations play a major role in the design and applicability of distributed concepts. A number of application domains have emerged which, by loosely coupling distributed components or utilizing latency hiding techniques, cope very well with latencies of distributed environments. Latency hiding techniques (such as caches, bulk operations, and interleaving of computation and communication) often require application level information to be effective (e.g. concurrency information of operations).

A library designed for distributed applications must allow these and other latency hiding techniques to be implemented.

2.7 End User Requirements

The SAGA specification was developed based on the responses to a call for use cases to the grid community [10, 11], and is designed to meet the resulting end user requirements.

An API implementation must meet other end user requirements outside the scope of the actual API specification, such as ease of deployment, ease of configuration, documentation, and support of multiple language bindings.

If any of these properties is missing, acceptance in the targeted user community will be severely limited.

3. GENERAL DESIGN

The implementation level requirements of the SAGA reference implementation as described in the previous section directly motivating 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 a 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.

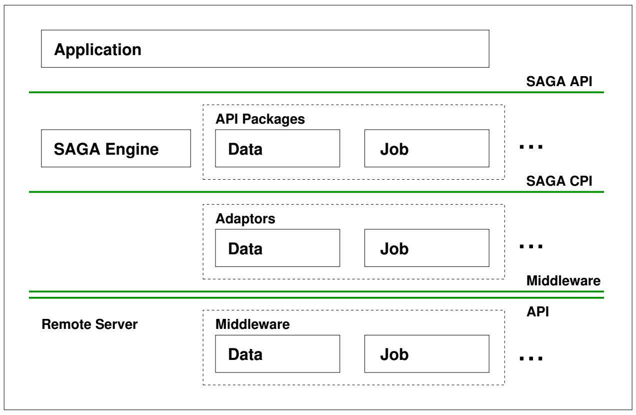

Figure 1: Architecture: A lightweight engine dispatches SAGA calls to dynamically loaded middleware adaptors. See text for details.

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.

As described above, a SAGA implementation must cope with a multitude of different dynamic requirements. 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, and immediately benefits from those.

3.2 The Overall Architecture

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

3.2.1 Horizontal Extensibility – API Packages

The SAGA specification is object oriented and defines a set of API groups keeping objects of related functionality together (packages). Our implementation uses this functional grouping to define *API packages*. Current packages are: file management, job management, remote procedure calls, replica management, and data streaming. Each of these packages constitutes a separate and independent module. These modules depend only on the SAGA engine, the user is free to use and link only those modules actually needed by the application, minimizing the memory footprint.

New API packages are expected to be added as the SAGA specification evolves. It is straightforward to add new packages 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:

- add the API (5) package files, and declare the classes,
- reflect the SAGA object hierarchy (more details below, in section 4.1.2),
- 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. The user then only adds the required include files for a fully fledged, compilable and usable SAGA API implementation package. This approach will also allow us to generate other SAGA language bindings from the SIDL specification, such as for C and FORTRAN.

We use the Boost.Wave [12] C++ preprocessor and special #pragmas it provides to pre-generate partially macro expanded sources. This overcomes the disadvantages of plain macros, simplifying debugging and improving readability.

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. Usually there will be a separate set of adaptors for each type of supported middleware. These adaptors implement a well defined *Capability Provider Interface* (CPI) and expose that to the top layer of the library, which makes it possible to switch adaptors at runtime and hence switch between different (and 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:

- 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;
- 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 [13] (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
- 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.

First, the private implementation classes all inherit from the same base class – only that base class is handled in the central engine module, so the engine can automatically cope with new API packages and adaptors. Second, all method calls are also handled generically in the engine.

The engine module is thus fully generic, and loosely coupled to both the API and adaptor layers. Any changes to the engine, all optimization, latency hiding techniques, monitoring features etc. can be implemented in the engine generically, and are orthogonal to the API and adaptor extensions. Hence, the extensibility of the engine represents the third orthogonal axis in the libraries extensibility scheme.

4. IMPLEMENTATION CHALLENGES AND 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[1]. That model prescribes the form of synchronous and asynchronous method calls. Essentially, each method call comes in three variants: as a *synchronous call*, as a *asynchronous call*, and as a *task call*. The synchronous call is, as expected, executed immediately. The asynchronous and task 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, i.e. the `run()` method was called on that task. For symmetry reason, we added a fourth version of method calls, which is again synchronous, but returns a `Finished` task. The C++ rendering of the SAGA task model is shown in figure 2.

SAGA task model

```
{
  using namespace std;
  using namespace saga;

  string src   = "any://host.net//data/src.dat";
  string dest1 = "any://host.net//data/dest1.dat";
  string dest2 = "any://host.net//data/dest2.dat";
  string dest3 = "any://host.net//data/dest3.dat";
  string dest4 = "any://host.net//data/dest4.dat";

  file f (src);

  // normal sync version of the copy method
  f.copy (dest1);

  // the three task versions of the same method
  task t1 = f.copy <task::Sync>  (dest2);
  task t2 = f.copy <task::ASync> (dest3);
  task t3 = f.copy <task::Task>  (dest4);

  // 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++

[1]The motivation for this task model is outside the scope of this paper, but is described in some details in [14]. This paper merely refers to those aspects relevant to the library design.

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[2], keeping it alive. This makes them very lightweight and copyable without major overhead, and therefore storable in any type of container.

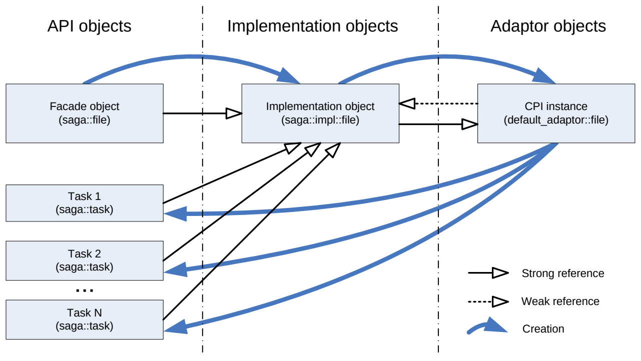

Figure 3: Object instance structure: Copying a API object instance means sharing state, returned tasks keep implementation alive.

As shown in figure 3, any API object instance creates the corresponding impl instance holding all the instance data of the SAGA object instance (those data defining the state of the API object instance, such as the name and current seek position of a file). Copying of a API instance therefore shares this state between the copied instances, which is probably what is expected by a user. Moreover, this behavior is consistent with anticipated handle based SAGA language bindings (such as for C or FORTRAN), where copying the handle representing a SAGA object instance naturally means sharing the internal instance data as well[3].

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 adaptor selection, instantiation, and creation of the required CPI object instance is implemented generically in the SAGA engine module and is used by all API packages. This process is injected into the API packages by the macros mentioned before (see section 3.2.1).

4.2 Inheritance and PIMPL

An interesting problem in the strict application of the PIMPL mechanism lies in the API object hierarchy: the

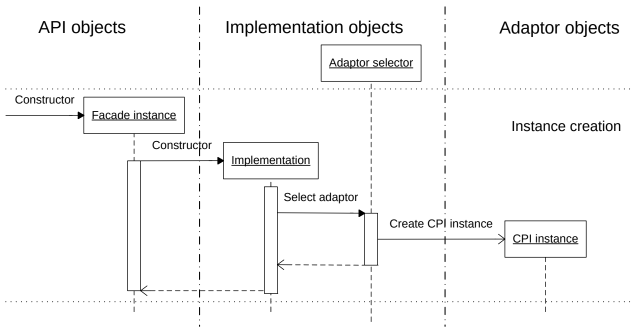

Figure 4: Object creation: Sequence diagram depicting the creation of all components as showed in figure 3. Note, how the call is intercepted by a SAGA engine module component to select a appropriate adaptor.

`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[4], and are built using single inheritance only, otherwise we would face object slicing problems when copying around the base classes only. To achieve this in our implementation:

- interfaces are added to the most derived classes by duplicating the interface functions, simplified by the usage of macros (interfaces as additional base classes would break the single inheritance) [5],
- the impl reference is down-casted and passed to the constructor of the respective base class (for example, the `saga::file` construction is shown in figure 5).

`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`

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

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

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

[3]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.

[4]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.

[5]The usage of macros for this isn't a problem, since, as mentioned above, these get pre-expanded during the build process.

API classes access the impl pointer through `get_impl()`, which, in derived classes, implies a static up-cast for the base class' impl pointer. As an example, the implementation of `get_impl()` for `saga::ns_entry` is shown in figure 6.

```
saga::ns_entry.get_impl()

boost::shared_ptr <saga::impl::ns_entry>
       ns_entry::get_impl (void) const
{
  // base class is saga::object
  return (boost::shared_ptr <saga::impl::ns_entry>
            (this->saga::object::get_impl (),
             boost::detail::static_cast_tag ()
        )  );
}
```

Figure 6: `get_impl()` implies a static cast of 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[6] inherits the `saga::impl::ns_entry` class, which inherits the implementation specific `saga::impl::proxy` class, which is 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. The mentioned state management in the PIMPL layers provides, as we have seen, for sharing state between separate API object instances. On a different layer, the adaptors represent operations on these 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[7] implemented by a single adaptor and corresponding to a single adaptor instance. These are naturally implemented by the adaptor writer as member data of the corresponding adaptor type.
- *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 most natural way were to implement this type of instance data as members of the corresponding CPI object. Unfortunately this is impossible since we cannot guarantee that the same CPI *instance* will get used for all API function calls on a particular API object instance (see section 4.4). For this reason these data are stored in a map in the impl object, identified by a universal unique identifier (UUID) [8].

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 must be carefully protected from race conditions potentially caused by the multithreaded nature of the overall implementation. Every adaptor needs to access at least one type of these instance data. Our implementation provides helper classes simplifying the correct locking of the instance data. Refer to figure 7 for an example of how to use these predefined wrappers for accessing the instance data members of a `saga::file` object. The main trick is that the wrapper classes implement a `operator->()` returning a pointer to the locked instance data. This lock is acquired during construction and is released during destruction of the wrapper instance.

Additionally, uniform state management is important to provide object state persistency in the future, with minimal impact on the existing code base.

```
Instance data type declaration

// file_instance_data can be used for the
// thread safe access to instance data
using namespace saga::adaptors;
typedef instance_data <file_cpi_instance_data>
    file_instance_data;
```

```
Instance data usage

void file_cpi_impl::sync_read ([args])
{
  // ... calculate 'bytes_read'
  {
    // the constructor aquires a lock
    file_instance_data data (this);

    // adjust instance data member 'pointer_'
    // (seek position) with number of bytes read
    data->pointer_ += bytes_read;

  } // lock goes out of scope here
}
```

Figure 7: Definition and use of a wrapper class to access instance data in a thread safe manner.

4.4 Generic Call Routing

We have already referred to the engines ability to generically route SAGA API method calls to adaptors. The essential idea of this routing mechanism is to represent these calls as abstract objects, and to redirect their execution depending on several attributes and the availability of suit-

[6]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`.

[7] For instance, the file and directory CPI are commonly implemented by one adaptor.

[8]All object instances in our SAGA implementation have an associated UUID allowing them to be uniquely identified.

able adaptors. 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 (the method gets executed in a thread then, making it asynchronous to some extent), or, if that is not available either, returns an error (`NotImplemented`).

This routing mechanism allows for

- trivial (synchronous) adaptor implementations,
- late binding: a different adaptor can be selected for each call, even on the same API object instance,
- variable adaptor selection strategies, e.g. based on adaptor meta data, user preferences and heuristics,
- latency hiding, e.g. by clustering related method calls (bulk optimization, see section 4.4.2), or by automatic load distribution over multiple adaptors (not implemented yet).

Figure 8 depicts the point in the sequence of calls where this call routing mechanism is injected by the SAGA engine.

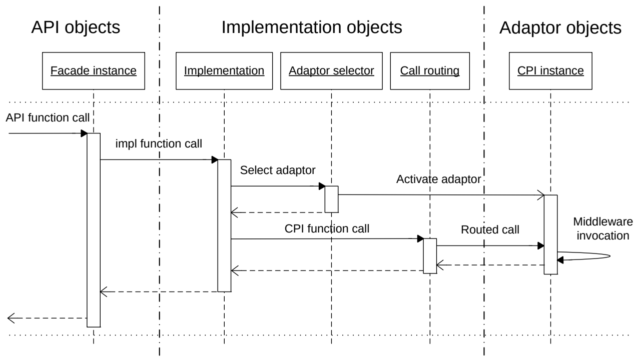

Figure 8: API function call: Diagram illustrating the execution sequence through the different object instances during a call to any adaptor supplied function.

4.4.1 *Sync/Async Routing*

As the SAGA API methods come in synchronous and asynchronous flavors (see section 4.1.1), adaptors would normally need to implement the methods in these two flavors as well. That, however, has been avoided by providing fallback implementations in the SAGA engine, if needed. The synchronous behavior is very easy to model: the asynchronous implementation has to be executed and waited for. The asynchronous behavior however requires the synchronous implementation to be wrapped into a thread. That thread then represents the asynchronous remote operation – internally, that thread is represented as a `saga::impl::task` instance. The realization of the `saga::impl::task` class bases on a implementation of the *futures* paradigm, a concurrency abstraction first proposed for MultiLisp [15].

It must be noted that this mechanism has a number of drawbacks: (a) the operation is not really asynchronous, as for example a dropped connection will likely cause it to fail; (b) the CPI instance executing the method still blocks (it is synchronous), which can, if badly implemented, cause locks on shared data structures; (c) the SAGA task model is, at some point in the future, to be extended, and tasks are then supposed to be able to survive an application life time – that would break the current implementation.

However, the mechanism allows simplifying adaptor implementations greatly, as most of the current existing grid middleware is *not* fully asynchronous anyway.

4.4.2 *Bulk optimizations*

Bulk optimizations represent a special form of latency hiding: multiple related, independent method invocations are clustered into a single call. That reduces the amount of remote communication needed to execute that method. A very common example is the execution of multiple remote I/O operations, which can be clustered in a single operation (the POSIX scattered I/O, `readv/writev (2)`, have a similar objective). Our implementation allows to combine tasks which are collectively run in a task container (see figure 9 for an example) to be clustered according to the method signature (e.g. same methods on the same object instance form one bulk), and can then passed to adaptors implementing bulk versions of that method [16]. If that bulk version is nowhere implemented, the methods are called one-by-one, as would be the default.

Usage of SAGA task_container

```
{
  using namespace std;
  using namespace saga;
  using namespace boost::assign;

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

  vector <string> dest;
  dest += "any://host.net//data/dest1.dat",
          "any://host.net//data/dest2.dat",
          "any://host.net//data/dest3.dat",
          "any://host.net//data/dest4.dat";

  // create a saga::task_container
  task_container tc;

  vector <string>::iterator end = dest.end   ();
  for (vector <string>::iterator it  = dest.begin ();
       it != end; ++it)
  {
    // add 'New' tasks to the task_container
    tc.add (f.copy <task::Task> (*it));
  }

  // run all tasks, then wait for all
  // bulk optimization is applied here.
  tc.run  ();
  tc.wait ();

  // all tasks are 'Finished' or 'Failed' now
}
```

Figure 9: Usage of the SAGA `task_container` class: This example illustrates, how the SAGA C++ implementation provides a simple and natural way to integrate grid related remote operations with well known C++ paradigms.

4.5 Adaptor Selection

The selection of suitable adaptors at runtime represents is a central component in the represented library implementation (see figure 8). It is, in general, 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 adaptors implementing that methods capability. All suitable adaptors are then ordered (best/most suitable first), and are tried one-by-one, until the method invocation succeeds. The adaptor selection again is routed through SAGA engine components, generically implementing this for any function to be routed to a CPI instance.

Now, that simple mechanism has a number of potential pitfalls. For one, as can be seen in figure 4, a number of adaptor instances (i.e. CPI instances) must be created. That can imply remote operations, and hence additional latencies. Secondly, the ordering of adaptors is very difficult, as it is hard to specify what constitutes a *good* or *suitable* adaptor. One metric is of course the availability of the required capability. Another utilized metric is the preference of adaptors providing the correct flavor (synchronous/asynchronous implementation).

Our library however allows adaptors to specify additional, key/value based meta data, and also allows to exchange the adaptor selection component. That way it is possible to (a) add additional meta data to adaptors (e.g. '`secure=yes/no`', or `'type=local/remote'`), and (b) add selection mechanism which evaluate and honor these meta data.

We apply a very simple optimization to the described scheme: if an adaptor was successfully invoked for an objects method call, the same adaptor is tried first on the next method call on the same object. That way, the adaptor selection is performed only once (on creation creation), and only repeated if any method invocation fails.

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[9]. We use macros in three different functions: for defining the API, for implementing API level interfaces, and for implementing the API on the implementation side of the PIMPL layer.

Figure 10 shows a part of the SAGA API definition in Scientific IDL (SIDL, [5]). The second part of the same figure shows the representation of the same SIDL segment in our implementation: the class depicted there is essentially complete! Adding a new API package can be done in minutes, and, in fact, is easy to automate (see section 3.2.1). The macros expand to all required flavors of the API (synchronous, asynchronous, task based, and task based synchronous – see section 4.1.1). The implementation macros retrieve the impl pointer via `get_impl()` (see sec 4.1.2), and invoke the respective impl method. Macros are also used to define and implement SAGA interfaces.

On the implementation side, the API is again specified and implemented by macros – these macros expand to implementations invoking the generic call routing described in section 4.4. That way, the impl classes are similarly thin and lightweight as the API itself. The examples in figure 10 do not show that the definition of the CPI are done by similar macros – these macros define an abstract base class, which

[9]As mentioned in section 3.2.1, we are using Boost.Wave features to pre-generate partially macro expanded sources to overcome the disadvantages of plain macros, hence simplifying debugging and improving readability.

`saga::file in SIDL`

```
class file : extends         saga::ns_entry,
             implements-all  saga::monitorable
{
  // ctor and dtor removed in this example
  is_file     (in  int            flags = None,
               out bool           test);
  read        (inout array<byte>  buffer,
               in    int          len_in,
               out   int          len_out);
  ...
}
```

`saga::file in C++ Macros`

```
// file.hpp
class file : public saga::ns_entry
{
  protected:
    boost::shared_ptr <impl::file> get_impl () const;

  private:
    PRIV_MONITORABLE;
    PRIV_0      (bool,    is_file, int);
    PRIV_2      (ssize_t, read,    char*, size_t);
    ...

  public:
    PUB_MONITORABLE;
    PUB_1_DEF_1 (bool,    is_file, int,   None);
    PUB_2_DEF_0 (ssize_t, read,    char*, size_t);
    ...
}

// file.cpp
IMPL_MONITORABLE;
IMP_1 (file, bool,    is_file, int);
IMP_2 (file, ssize_t, read,    char*, size_t);
...
```

`saga::impl::file in C++ Macros`

```
// impl/file.hpp
class file : public saga::impl::ns_entry
{
  public:
    IMP_DECL_MONITORABLE;
    IMP_DECL_1 (bool   , is_file, int);
    IMP_DECL_2 (ssize_t, read,    char*, size_t);
    ...
}

// impl/file.cpp
IMP_IMPL_MONITORABLE;
IMP_IMPL_1 (file, file_cpi, bool,    is_file, int);
IMP_IMPL_2 (file, file_cpi, ssize_t, read,    char*,
            size_t);
...
```

Figure 10: Macro based API definition and implementation (macro names abbreviated)

is then implemented by the adaptors, essentially implementing the original SAGA API, and providing the required grid capabilities.

Our implementation uses macros in well defined locations, and they allow for simple extensibility of the API. In fact, we consider the usage of our SAGA implementation to implement other APIs for distributed systems, and also to re-implement earlier grid APIs for backward compatibility (such as GAT) – the macros as shown, and the generic call routing, make that a very simple exercise.

5. LESSONS LEARNT – IMPLEMENTATION PROPERTIES

Thus far, this paper has motivated the design objectives of the SAGA C++ Reference implementation, and described several implementation techniques used to meet these objectives. This section will summarize the resulting properties of the SAGA implementation from an end user perspective, and motivate further developments and extensions.

5.1 Uniformity over Programming Languages

The SAGA API specification is language independent – however the goal is to define language bindings which provide both a language-native look & feel to the API user, and strive for syntactic and semantic similarity over all SAGA language bindings. One of the consequences of this goal is that the API specification 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, to provide the same semantics, and similar look & feel to other language bindings. A Python wrapper for our library is in alpha status, and we plan to add similar thin wrappers to provide bindings to C, FORTRAN, Perl, and possibly others.

From another point of view, we find it extremely convenient to be able to implement *adaptors* in different languages. The Grid Application Toolkit (GAT, [3]), a C-based API predecessor of SAGA, already allows adaptors in different languages, and we may implement similar mechanisms to allow Python or C based adaptors for this SAGA implementation as well. In particular Python based adaptors have been extremely useful for rapid prototyping of middleware bindings for GAT.

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

The dynamicity of grid middleware has already been mentioned, as a central dominating property of grid environments. This is addressed in our implementation by the described adaptor mechanism which binda to diverse middleware. Additionally, late binding, fall back mechanisms, and flexible adaptor selection allow for additional resilience against an dynamic and evolving run time environment. It should be noted, however, that 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 – our SAGA implementation usability will be severely impacted if the quality of adaptors undermines the libraries mechanisms.

5.3 Modularity makes the Implementation Extensible

Section 4.6 described how the SAGA implementation will be able to cope with the expected evolution and extension of the SAGA API. Further, the adaptor mechanism allows for easy extensions of the library, to provide additional middleware bindings. In fact, the major future work for our SAGA implementation will be to provide multiple sets of stable adaptors for the major grid environments. We expect, however, that this task requires massively more effort than the implementation of the presented library, and we hope that grid middleware vendors will be motivated to support and maintain these adaptors. Ideally, middleware vendors will *implement* adaptors for SAGA, and deliver them as part of their client side software stack in the same way they provide MPI implementations. This would be a major step towards wide spread grid applications.

5.4 Portability and Scalability

Heterogeneous distributed systems naturally require portable code bases. We think that our library implementation is in fact very portable, as we strictly adhere to the C++ standard and portable libraries. In fact, we currently develop the library on Windows and Linux concurrently, so we are confident that we are able to cover the two major target platforms without any problems – but we don't expect (and currently don't encounter) any problems on other platforms. It must be noted, however, that 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. As SAGA introduces a number of communication mechanisms, scalability concerns are naturally also raised in respect to SAGA implementations. First, the SAGA API is not targeting high performance communication schemes, but tries to stick to simple communication paradigms – in no sense does SAGA intend to replace MPI or other distributed communication libraries. Having said that, our design allows for zero-copy implementations of the SAGA communication APIs, and also allows 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 to use. However, simplicity of use 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.

It is a challenge to keep a library implementation, such as this one, ifself simple, with readable code. Again, a modular approach helps here. For example, it is simple to hide the generic call routing, or the adaptor selection, in the engine module, as these features are not usually exposed to the user or adaptor programmer. However, we believe that modeling these central properties as modules increases the readability and maintainability of the code significantly.

The SAGA API implicitly introduces a concurrent programming model, due to its notion of asynchronous operations, or tasks. The C++ language binding of the API, and our implementation, allows to combine that model with arbitrary mechanisms for managing concurrent program elements (i.e. to ensure object state consistency in all circumstances, to ensure thread safety, and to allow for application level semaphores and mutexes).

6. FUTURE WORK

As mentioned, work on appropriate middleware adaptors will undoubtedly require significant resources in the future – but without those, the SAGA API will not be usable in real grid environments. This motivates us to work now on simplifying adaptor creation, integration and maintenance, and seek support and contributions from the OpenSource community, and from grid middleware vendors. We deem adaptor development and support to be now more important than the API development itself.

In parallel, we will develop other language bindings for our implementation, as motivated in section 5.1, and the mechanisms to allow adaptors in various programming languages. Finally, we plan to apply further generic latency hiding techniques, and to experiment with other API implementations in our framework.

7. CONCLUSION

We have described the C++ reference implementation of the SAGA API, which is designed as a very generic and extensible framework: it allows for very simple extension of the SAGA API (and in fact is easily usable for other APIs); it allows for run-time extension of middleware bindings, which is essential for nowadays grid environments; and it allows for orthogonal optimizations and features, such as late binding, diverse adaptor selection strategies, and latency hiding techniques. We described the used techniques enabling these features, amongst them the application of the PIMPL paradigm for a complete class hierarchy, generic call routing, and extensive use of C++ preprocessor macros for API definitions.

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 to be learned is that distributed environments *allow* for fancy mechanisms, which would be too expensive in local environments. Fail safety and latency hiding mechanisms are enormously more important than, for example, virtual functions, late binding, and additional abstraction layers. In that sense, we hope that the presented library is of use to other implementors of distributed applications.

The source code of this library is freely available under the Boost license, and can be accessed via anonymous CVS - details can be found on `http://wiki.cct.lsu.edu/saga/`.

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 our work.

9. ADDITIONAL AUTHORS

Gabrielle Allen
Center for Computation & Technology
Louisiana State University
Baton Rouge, Louisiana, USA
email: `gallen@cct.lsu.edu`

10. REFERENCES

[1] SAGA Core Working Group. Simple API for Grid Applications – API Version 1.6. 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] Gabrielle Allen, Kelly Davis, Tom Goodale, Andrei Hutanu, Hartmut Kaiser, Thilo Kielmann, Andre Merzky, Rob van Nieuwpoort, Alexander Reinefeld, Florian Schintke, Thorsten Schütt, Ed Seidel, and Brygg Ullmer. The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid. *Proceedings of the IEEE*, 2004.

[4] Dayong Huang and Gabrielle Allen and Chirag Dekate and Hartmut Kaiser and Zhou Lei and Jon MacLaren. getdata: A Grid Enabled Data Client for Coastal Modeling. In *High Performance Computing Symposium (HPC 2006)*, April 3–6 2006.

[5] Babel Project. Scientific Interface Definition Language (SIDL). `http://www.llnl.gov/CASC/components/babel.html`.

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

[7] Open Grid Forum (OGF). `http://www.ogf.org/`.

[8] OGSA Working Group. Defining the Grid: A Roadmap for OGSA Standards. Technical report, Open Grid Forum, September 2005. GFD.53.

[9] The Globus Alliance. `http://www.globus.org/`.

[10] Andre Merzky and Shantenu Jha. A Requirements Analysis for a Simple API for Grid Applications. Technical report, Open Grid Forum, May 2006. GFD.71.

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

[12] Hartmut Kaiser. Boost.Wave: A Standard Compliant C++ Preprocessor Library. `http://www.boost.org/libs/wave/index.html`.

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

[14] Tom Goodale, Shantenu Jha, Hartmut Kaiser, Thilo Kielmann, Pascal Kleijer, Gregor von Laszewski, Craig Lee, Andre Merzky, Hrabri Rajic, and John 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.

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

[16] Stephan Hirmer, Hartmut Kaiser, Andre Merzky, Andrei Hutanu, and Gabrielle 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.