

## The SAGA C++ API Programmer's Guide

The Simple API for Grid Applications (SAGA) is a collection of C++ API packages defined on top of an overall programmatic appearance. The SAGA API packages try to cover the bulk of the requirements needed for grid programming like, for example, job submission, remote file transfer and replica management, to name a few. This guide aims for the developer who wants to learn the basics of how to use SAGA and C++ to develop applications that can make use of a broad set of existing distributed and grid computing infrastructures.<sup>1</sup>

### Status of This Document

This guide is still work in progress.

---

<sup>1</sup>editor

<sup>1</sup>In order to understand the intent and also the limitations of the SAGA approach, it is useful to read the Wikipedia entry on "Leaky Abstractions" ([http://en.wikipedia.org/wiki/Leaky\\_abstraction](http://en.wikipedia.org/wiki/Leaky_abstraction)), and also Joel Spolsky's column on the same topic (<http://www.joelonsoftware.com/articles/LeakyAbstractions.html>).

## Contents

## 1 Introduction

What are Grids? How can you use them? Why would you need an API for doing so? What makes SAGA different from other Grid APIs?

These are some of the questions which we will try to answer in this introduction. Don't expect a complete treatise on the subject though; in stead, we will refer the interested reader to the literature where appropriate. Also, this introduction can safely be skipped by readers which are familiar with Grid programming, or even just with using Grids.

The text assumes that the reader has some familiarity with the concept of distributed computing and programming of distributed applications.

### Grid Computing

In “What is a Grid?” [?], a three point checklist is provided. Although by no means rigorous or complete, it provides a starting point. Following that, Grids are:

1. coordinated resources that are not subject to centralized control,
2. use standard, open, general-purpose protocols and interfaces,
3. deliver non-trivial qualities of service.

### Grid Standardization

A diverse Grid Computing community has emerged over the past few years; the Open Grid Forum (OGF) [?] serves as a focus point for the Grid community groups and Grid standardization efforts. The main topics of standardization are many fold, and cover, amongst others,

- service architectures
- service interfaces
- wire protocols
- information models
- Grid related markup languages
- user interfaces

For this document, as might be obvious, the last point, which includes Application Programming Interfaces (APIs), is of special interest. It should be

noted that OGF's standardization mostly targets Grid middleware developers and Vendors, and that the OGF's API specifications are the top layer of OGF's standards stack.

## Grid APIs

Traditionally, most distributed (and Grid) middleware systems come with a 'native' API. Best known for such an API is Globus, a pioneering Grid Middleware project: each version of Globus services was/is accompanied by an extensive API which allows programmers (end-user application or otherwise), to make use of these services, and provides access to Globus service client libraries, Globus protocol implementations, Globus security tools, Globus markup languages, etc.

Despite the success of Globus, its approach exhibits a number of problems: applications written against a specific version of the Globus API are not (easily) portable to other versions of the API, and not portable at all to other Grid middleware, such as Genesis-II, GridSam, or Unicore. Furthermore, typical middleware APIs expose a specific set of features available in that middleware. These do not necessarily match the Grid abstractions required by the various applications; higher level services which provide these abstractions, are then required to implement their own specific APIs.

That situation is, to some extent, comparable to the 80's and 90's, where many vendors of parallel computers shipped their specific proprietary communication library along with their product, to enable the end users to make use of the distributed compute power from within a single (distributed) application. Before the establishment of MPI as a single dominant standard for this inter-node communication, the creation of *portable* distributed applications was an extremely difficult and tedious effort. Only the adoption of MPI by basically all cluster vendors, and the *native support* for MPI on most platforms, could finally alleviate that problem<sup>2</sup>.

## SAGA

The Simple API for Grid Applications (SAGA), a proposed OGF standard, is in several ways a comparable effort to MPI: SAGA tries to establish a *single* API for Grid application programmers, which is ideally shipped with all Grid middleware, so that the programmer can focus on the application logic, instead of having to deal with tedious and complex middleware details.

---

<sup>2</sup>Note that MPI serves a specific problem space of tightly coupled massive parallel applications. Loosely coupled parallel applications, for example, are served by a different set of distributed communication concepts, e.g., Peer2Peer systems, or component models, etc.

But SAGA tries to go further: instead of defining a fixed single common denominator, SAGA is by definition an extensible and modular effort. By being extensible both horizontally and vertically, SAGA should be able to adapt to a variety of use cases and user communities. Additionally, our SAGA implementation can be extended by additional middleware bindings, to support the widest possible portability for applications.

### Horizontal Extensibility of SAGA

The SAGA approach is extremely modular: a stable and finite core set of SAGA Look & Feel packages is accompanied by a variable set of functional API packages. That latter set is expected to grow over time, and to cover future emerging Grid programming paradigms.

#### **FIXME:** **graphics goes here**

At the moment, the SAGA API covers the following functional packages:

<b>jobs:</b>	job creation and management
<b>files:</b>	interaction with file systems
<b>replica:</b>	management of logical files
<b>streams:</b>	BSD socket oriented IPC
<b>rpc:</b>	remote procedure calls

All these packages are provided by our SAGA implementation. Additional packages, which are in the process of being defined, include:

<b>advert:</b>	persistent storage of application level information <sup>2</sup>
<b>sd:</b>	service discovery <sup>2</sup>
<b>messages:</b>	message based IPC
<b>cpr:</b>	checkpoint and recovery
<b>dais:</b>	database access and integration

### Vertical Extensibility of SAGA

The experiences in various OGF user communities show that the Grid programming models used in different application domains vary widely. In particular, it seems impossible to completely standardize information models, data models, and data formats, without losing the abstractive power of a high level application oriented API.

---

<sup>2</sup>These packages are also provided in our implementation.

**FIXME: graphics goes here**

For that reason, SAGA tries to stay independent of these issues, and, at the same time, offers (a) the flexibility to natively accommodate a variety of data types and structures, and (b) the ability to additionally define higher level API packages, which address application domain specific services and programming models. If and how these additional high level packages are standardized is then up to the community creating these packages. SAGA, however, maintains a uniform and consistent API for a very wide variety of users and use cases, without being too specific and inflexible.

**Implementation Extensibility**

Any Grid API is only as good and powerful as its underlying middleware is. How does this statement hold for SAGA, which is by design not bound to a specific Grid middleware? It holds true in fact: although the syntax and semantics of the SAGA API calls remain stable over the various Grid middlewares SAGA is running on top of, the specific performance characteristics, and in fact the exact set of SAGA calls provided, may vary from case to case.

That may seem like a serious drawback, and probably is, but an API can do only so much to emulate missing middleware features. In any case, to simplify the runtime portability of applications, our SAGA implementation allows binding to a variety of Grid middlewares *at runtime*, i.e., without the need to even relink your application.

**FIXME: graphics goes here**

That functionality is provided by a plugin type mechanism, which loads middleware *adaptors* into SAGA as required. These adaptors translate the SAGA API calls to the specific Grid middleware actions. For more details on adaptors and their maintenance and configurations, please refer to the installation manual.

**1.1 Getting Started**

The reader should by now have a fair idea of the target scope of the SAGA API: it is designed to simplify the programming of novel applications which are to be run on a variety of Grid middlewares.

The next chapter, the 'Quick Start Guide', should be enough to get you going, and to compile and run small SAGA applications. The chapters after that will discuss the SAGA Look & Feel in more detail, and also dive into the various functional packages we provide. The chapters are independent, no specific reading order is implied, unless noted otherwise.

## 2 Quick Start Guide

So, you are one of the impatient readers who dread long dry user manuals? No problem: read this chapter, and you will have a good starting point to use SAGA. After all, the **S** in **SAGA** stands for **Simple**! The remainder of the document provides considerable more details about the API, but you can defer the later chapters until you have indeed the need for more information.

It is important to understand that the API consists of two parts: the Look & Feel, and the API packages. The packages are what you are most likely interested in, because they provide the means to interact with the Grid: they start jobs, copy files, perform remote procedure calls, etc. The Look & Feel provides the syntactic and semantic expressiveness to control *how* these actions are expressed (syntax) and performed (semantics). For example, the session management in the Look & Feel part tells you how to specify the security constraints of your actions, while the task model from the Look & Feel determines how you can express synchronous versus asynchronous actions.

That all sounds rather theoretical: let us dive into some examples! The first one allows you to copy a file<sup>3</sup>:

File copy

```
#include <saga/saga.hpp>

int main (int argc, char** argv)
{
    // do a file copy
    saga::url u(argv[1]);
    saga::filesystem::file f (u);
    f.copy (saga::url (argv[2]));
}
```

Isn't that simple? Three lines, and your file is copied! `file.copy()` is provided by the `file` class in the `saga::filesystem` package. That is a *functional* SAGA call. So, how will *non-functional* properties of that call enter the picture? Let's try to have the same call asynchronously:

File copy (async version)

```
#include <saga/saga.hpp>

int main (int argc, char** argv)
{
    // run a file copy asynchronously
}
```

---

<sup>3</sup>For the sake of brevity, we leave error and sanity checks out of the examples

```
saga::url u(argv[1]);
saga::filesystem::file f (u);
saga::task t = f.copy <saga::task::Async> (saga::url (argv[2]));

// do something else

// wait for the copy task to finish
t.wait ();
}
```

That is almost the same. In particular, the signature for the `file.copy()` method is the same. It is, however, now qualified as `saga::task::Async`, and returns a `saga::task` instance. That (stateful) task represents the asynchronous call. You can think of the synchronous qualification as the default, which can be left out.

Another Look & Feel package deals with monitoring, and allows, for example, to get notifications if the task finishes. Yet another Look & Feel package allows to specify security credentials for the copy operation, and so on and so forth.

So, that is the general picture: SAGA packages provide you with means to interact with the Grid environment, and the Look & Feel packages determine how these actions are executed, syntactically and semantically, altering the functional method's performance.

The remainder of the document will give more details, first about the Look & Feel packages of SAGA, and then about the functional packages. Each section starts with a quick introduction (which should be enough to get you going in most cases), followed by a reference section, and by a more detailed discussion where appropriate. Finally, a number of advanced topics are discussed, which are probably not of your interest initially, but they are certainly useful concepts as soon as your application reaches a certain complexity itself.

Before diving into the API itself, we will shortly describe how to compile and run SAGA applications, so that the reader will be able to actively follow the coding examples and exercises.



## 3 Building and Running your Application

### 3.1 Building your Application

For Unix-like systems, a configure/make based build system is provided. That system can also be used to compile your SAGA application, and is (briefly) described below.

Furthermore, your SAGA installation offers a tool named **saga-config**, which can also be used to determine the relevant compile parameters.

SAGA uses a gnu-make based build system. It includes a number of makefiles throughout the source tree, and the `$SAGA_ROOT/make/` directory. All these *make includes*<sup>4</sup> get installed into `$SAGA_LOCATION/share/saga/make/`<sup>5</sup>.

When compiling your application, these make includes may provide an easy starting point. All you need to do is to set `SAGA_LOCATION` to have it point to your SAGA installation root, and to include `saga.application.mk`, which will add all rules required to build a SAGA application. The following Makefile stub should get you started:

```
----- Makefile for SAGA applications -----  
  
SAGA_SRC  = $(wildcard *.cpp)  
SAGA_BIN  = $(SRC:%.cpp=%)  
  
include $(SAGA_LOCATION)/share/saga/make/saga.application.mk
```

This stub loads the make rules, etc., needed to build the application. If the application needs additional include directories or libraries, use the following syntax *after* the make includes:

```
----- Makefile: setting compiler/linker flags -----  
  
SAGA_CPPFLAGS += -I/opt/super/include  
SAGA_LDFLAGS  += -L/opt/super/lib -lsuper
```

Of course it is possible to build SAGA applications with custom Makefiles. The SAGA make includes can still, however, be used to obtain the SAGA specific compiler:

---

<sup>4</sup>A make include is a makefile building block

<sup>5</sup>For details on building and installing SAGA, please refer to the SAGA installation manual.

---

Custom Makefile

---

```
SRC      = $(wildcard *.cpp)
OBJ      = $(SRC:%.cpp=%.o)
BIN      = $(SRC:%.cpp=%)

CXX      = g++
CXXFLAGS = -c -O3 -pthread -I/opt/mpi/include

LD       = $(CXX)
LDFLAGS  = -L/usr/lib/ -lc

include $(SAGA_LOCATION)/share/saga/make/saga.engine.mk
include $(SAGA_LOCATION)/share/saga/make/saga.package.file.mk

all: $(BIN)

$(OBJ): %.o : %.cpp
        $(CXX) $(CXXFLAGS) $(SAGA_CXXFLAGS) -o $@ $<

$(BIN): % : %.o
        $(LD) $(LDFLAGS) $(SAGA_LDFLAGS) -o $@ $<
```

SAGA\_CXXFLAGS and SAGA\_LDFLAGS contain only those options and settings which are required to use SAGA. You may want to use 'make -n' to print what the resulting make commands are, in order to debug eventual incompatibilities between the SAGA compiler and linker flags, and your own ones.

Yet another option is to use the output of saga-config for makefiles:

---

saga-config used in Makefile

---

```
SRC      = $(wildcard *.cpp)
OBJ      = $(SRC:%.cpp=%.o)
BIN      = $(SRC:%.cpp=%)

CXX      = gcc
CPPFLAGS = -I/opt/mpi/include
CXXFLAGS = -c -O3 -pthread

LD       = $(CXX)
LDFLAGS  = -L/opt/mpi/lib/ -lmpi

CPPFLAGS += $(shell $(SAGA_LOCATION)/bin/saga-config --cppflags)
CXXFLAGS += $(shell $(SAGA_LOCATION)/bin/saga-config --cxxflags)
LDLAGSS  += $(shell $(SAGA_LOCATION)/bin/saga-config --ldflags)

.default: $(BIN)
```

```
$(BIN): % : %.cpp

$(OBJ): %.o : %.cpp
    $(CXX) $(CPPFLAGS) $(CXXFLAGS) -o $$@ $<

$(BIN): % : %.o
    $(LD) $(LDFLAGS) -o $$@ $<
```

## Linkage Options

The SAGA libraries come in two flavours: *standard* and *lite*. The standard version is what you have observed in the examples above: your application is linked against the `libsaga_engine`, and against the available/required packages (`libsaga_package_abc`). The adaptor libraries (`libsaga_adaptor_xyz`) are not linked to the application, but are loaded at runtime.

That standard version offers the most flexibility for your application, and allows it to adapt linkage to a wide range of use cases. The runtime adaptor loading allows you to adapt your application to Grid middleware variations at runtime. That comes at a cost: at runtime, the shared library dependencies have to be resolved, and SAGA has to be correctly configured to find the adaptor libraries at runtime. The former can be resolved by linking your application statically, but the SAGA runtime configuration cannot be avoided.

That is the reason why we provide the second, 'lite' version of the SAGA library: it is a *single* shared library which contains the SAGA engine, all available packages, and a set of adaptors. These adaptors are thus *not* loaded at runtime, but are linked at link-time. Other adaptors can, however, still be loaded at runtime, as before.

That way, no SAGA runtime configuration is required at all, as all dependencies are resolved on link-time. The SAGA installation manual provides more information on the selection of adaptors to be included into the (`libsaga_lite`). Note that this comes at a cost as well: your application binary will be larger, as all the adaptors are linked against it, whether they are needed or not.

The SAGA make system supports linking against the (`libsaga_lite`) like this:

```
----- Makefile for SAGA-Lite -----

SAGA_SRC  = $(wildcard *.cpp)
SAGA_BIN  = $(SRC:%.cpp=%)

SAGA_USE_LITE = yes
```

```
include $(SAGA_LOCATION)/share/saga/make/saga.application.mk
```

## 3.2 Running your Application

Your SAGA application should have only few runtime dependencies. First of all, it needs to find the shared libraries you used, and the shared libraries your SAGA installation is linked against. That may include libraries required by the SAGA adaptors, e.g., globus libraries, openssl, etc. Depending on your operating system, you may need to set the environment variables `LD_LIBRARY_PATH`, `DYLD_LIBRARYPATH`, or similar.

After starting, SAGA reads a couple of initialization files to determine your specific set of adaptors, and several configuration options for these adaptors. In general, you should not have the need to touch these ini files – SAGA configuration and installation is supposed to be performed by the system administrator, not the end user. In most cases, SAGA should be able to find these ini files automatically (the installation prefix is statically compiled into SAGA. If that fails, SAGA searches the following locations (in this order) for the main `saga.ini` file:

```
/etc/saga.ini
$SAGA_LOCATION/share/saga/saga.ini
$HOME/.saga.ini
$PWD/.saga.ini
$SAGA_INI
```

Those ini files point to the individual adaptor ini files, which then allow SAGA to load these adaptors.

While running, the SAGA library can print log messages, of varying verbosity. That output is controlled by the environment variable `SAGA_VERBOSE`. The values are as follows:

```
1: Critical
2: Error
3: Warning
4: Info
5: Debug
6: Blurb
```

When `SAGA_VERBOSE` is not set, the library is silent.

## Part I

# General SAGA Concepts

A number of concepts and paradigms are used in all packages of the SAGA API. The SAGA specification names these concepts as the '*SAGA Look & Feel*'. This section describes the most important concepts.

As this section does not actually provide the reader with the means to perform any useful remote operation in Grids, the reader may want to skip this part and continue to read Part ??, and come back to this part as needed.

## 4 Error Handling

### 4.1 Quick Introduction

SAGA error handling is exception-based: a rather flat hierarchy of 14 exception types inherit the properties of the general `saga::exception` class. That allows to inspect the exception for its exact type, and for the detailed error message(s) associated with this exception.

**HINT:**

As our SAGA implementation is adaptor-based, a single API call could actually return more than one exception at the same time. That is rendered by returning the most specific exception, whose error message then also contains the error messages from all the other exceptions, usually one per adaptor.

### 4.2 Reference

Prototype: `saga::exception` class and derivatives

```
namespace saga
{
class exception : public std::exception
{
public:
    ~exception() throw() {}

    // Gets the message associated with the exception
    char const* get_message() const throw()

    // an alias for get_message
    char const* what() const throw()

    // get type of exception
    saga::error get_error () const

    // Gets the SAGA object associated with exception.
    saga::object get_object () const throw()
};

// parameter related exceptions
class parameter_exception : public saga::exception
```

```
class incorrect_url      : public saga::parameter_exception
class bad_parameter      : public saga::parameter_exception

// state related exceptions
class state_exception    : public saga::exception
class already_exists     : public saga::state_exception
class does_not_exist     : public saga::state_exception
class incorrect_state    : public saga::state_exception
class timeout            : public saga::state_exception

// security related exceptions
class security_exception : public saga::exception
class permission_denied  : public saga::security_exception
class authorization_failed : public saga::security_exception
class authentication_failed : public saga::security_exception

// general exceptions
class no_success         : public saga::exception
class not_implemented    : public saga::exception
```

### 4.3 Details

Error handling in SAGA is completely exception-based<sup>6</sup>. The following exceptions exist (note that the exception class names and the `saga::error` enums, returned by `exception.get_error()` have identical names):

#### NotImplemented

A method is specified in the SAGA API, but is not provided by this specific SAGA implementation<sup>7</sup>.

#### IncorrectURL

This exception indicates that a URL argument could not be handled. For example, this implementation may be unable to handle the specified protocol, or the access to the specified entity, via the given protocol, is impossible.

#### BadParameter

This exception indicates that at least one of the parameters of the method call is ill-formed, invalid, out of bounds or, otherwise, not usable. The

<sup>6</sup>The SAGA specification allows for an additional `error_handler` interface to be implemented by SAGA objects—that seemed unnecessary for object-oriented languages such as C++.

<sup>7</sup>In particular, the method is not provided by any of the available adaptors, see Section ??.

error message gives specific information on what parameter caused the exception, and why.

**AlreadyExists**

This exception indicates that an operation cannot succeed because an entity to be created already exists, and cannot be overwritten. Explicit flags on the method invocation may allow the operation to succeed, e.g., if they indicate that Overwrite is allowed.

**DoesNotExist**

This exception indicates that an operation cannot succeed because a required entity is missing. Explicit flags on the method invocation may allow the operation to succeed, e.g., if they indicate that Create is allowed.

**IncorrectState**

This exception indicates that the object a method was called upon is in a state where that method cannot possibly succeed. A change of state might allow the method to succeed with the same set of parameters.

**PermissionDenied**

An operation failed because the identity used for the operation did not have sufficient permissions to successfully perform the operation.

**AuthorizationFailed**

An operation failed because none of the available contexts of the used session could be used to access the given resource. In contrast to the **PermissionDenied**, this exception usually indicates an error on the administrative level, which usually cannot be fixed by the end user.

**AuthenticationFailed**

An operation failed because none of the available session contexts could successfully be used for authentication. That exception usually indicates invalid or outdated security credentials.

**Timeout**

This exception indicates that a remote operation was not completed successfully, because the network communication or the remote service timed out. This exception is never thrown if a timed `wait()` or similar method times out, as that is not an error condition.

**NoSuccess**

This exception indicates that an operation failed for any other reason. The exception message may, or may not, contain some more details about the cause of the error.



Even for simple SAGA operations, the implied Grid interactions can be fairly complex and may invoke multiple remote operations. It may thus happen that multiple exceptions apply during the execution of the method. In such cases, the **most specific exception** is thrown (the list of exceptions above is ordered, with the most specific exception up front).

Also, as the above Reference section shows, the exceptions are ordered in an exception hierarchy. The application can thus try to catch whole classes of exceptions, e.g., all security related exceptions, or all state related exceptions. The code example below shows, however, how the lower level exceptions can be accessed for debugging purposes:

Code Example

```
try
{
    saga::url u ("/path/which/does/not/exist");
    saga::filesystem::file f (u);
}
catch ( saga::state_exception const & e )
{
    switch ( e.get_error () )
    {
        // handle does not exist
        case saga::DoesNotExist:
        {
            std::cout << "file does not exist"
                      << std::endl;

            exit (-1);
            break;
        }

        // generic handler for state related problems
        default:
        {
            std::cout << "some other saga state exception caught: "
                      << std::string (e.what ())
                      << std::endl;

            exit (0);
            break;
        }
    }
}
```

## 5 Using Data Buffers

### 5.1 Quick Introduction

Various classes (e.g., `saga::file` and `saga::stream`) in the SAGA API expose I/O operations, i.e., chunks of binary data can be written to, or read from these classes. Other classes (such as `saga::rpc`) handle binary data as parameters. In order to unify the application management of these data, SAGA introduces the `saga::buffer` class, which is essentially a simple container class for a byte buffer, plus a number of management methods. Various subclasses of the `saga::buffer` exist, and, as described below, users are allowed, and actually encouraged, to build their own ones.

The C++ rendering of SAGA distinguishes between mutable and non-mutable buffers: non-mutable buffers are used for write-type operations, and cannot be changed by the SAGA implementation; mutable buffers are for read-type operations, and can be changed (i.e., new data can be added to the buffer).

### 5.2 Reference

Prototype: `saga::buffer`

```
namespace saga
{
    class const_buffer
        : public saga::object
    {
    public:
        const_buffer (void const * data,
                      saga::ssize_t size);
        ~const_buffer (void);

        saga::ssize_t  get_size (void) const;
        void const *   get_data (void) const;
        void           close   (double timeout = 0.0);
    };

    class mutable_buffer
        : public saga::const_buffer
    {
    public:
        typedef void buffer_deleter_ type(void* data);

        typedef TR1::function <buffer_deleter_type> buffer_deleter;
        static void default_buffer_deleter (void * data);
    };
}
```

```
mutable_buffer (saga::ssize_t size = -1);
mutable_buffer (void * data,
               saga::ssize_t size);
mutable_buffer (void * data,
               saga::ssize_t size,
               buffer_deleter cb);
~mutable_buffer (void);

void set_size (saga::ssize_t size = -1);
void set_data (void * data,
               saga::ssize_t size,
               buffer_deleter cb = default_buffer_deleter);
void * get_data (void); // non-const version
};
}
```

## 5.3 Details

Although the concept of an I/O buffer is very simple, and the prototype shown above is rather straight forward, the semantic details of the SAGA buffer are relatively rich. That holds true in particular for the memory management of the buffer data segment. For the interested reader, the `saga::buffer` section in the SAGA Core API specification contains quite some detail on that issue.

### 5.3.1 Buffer Memory Management

In general, buffers can operate in two different memory management modes: the data segment can be user-managed (i.e., application-managed), or SAGA-managed (i.e., implementation-managed). The constructors allow the application to pass a memory area on buffer creation: if that buffer is given, and not-NULL, then the SAGA implementation will use that buffer, and will never re- nor de-allocate that memory (memory management is left up to the application). On the other hand, if that memory area is not given, or given as NULL, then the SAGA implementation will internally allocate the required amount of memory, which **MUST NOT** be re- or de-allocated by the application (memory management is left to the SAGA implementation).

Although the latter version is certainly convenient for the end user, it comes with a potential performance penalty: data from the implementation allocated buffer may sometimes need an additional memcpy into application memory. If that is the case, it is up to you to decide what memory management mode works best for your application use case.

**HINT:**

The most performant case is most of the time to re-use a single (or a small set of) application allocated memory buffer(s) over and over again. Note that you can use a larger size memory segment for a small buffer by giving a smaller size parameter to the constructor.

---

Example `saga::buffer` usage

---

```
// allocate a 'large' buffer statically
char mem[1024];

// create a saga buffer object for reading (mutable)
saga::mutable_buffer buf (mem, 512);

// open a file
saga::url u ("/etc/passwd");
saga::filesystem::file f (u);

// read data into buffer - the first 512 bytes get fill
f.read (buf);

// seek the buffer, so that the next read goes into the
// second half of the buffer
buf.set_data (mem + 512, 512);

// now read again
f.read (buf);

// the complete buffer should be filled now
// print what we got
std::cout << mem << std::endl;
```

### 5.3.2 Const versus Mutable Buffers

On **write**-like operations, the SAGA implementation has no need to change the buffer's data segment in any way: it only needs to read the data, and to copy them to whatever entity the write operations happens upon. The implementation can thus treat the buffer as `const`, which allows a number of optimizations and memory access safeguards to be employed.

On the other hand, **read**-like operations will usually require the SAGA implementation to write, or even to (re-)allocate the buffers memory segment. In such cases, `const` safeguards cannot be employed.

**HINT:**

It is encouraged the use of `const_buffer` instances for `write`-like operations, and of `mutable_buffer` instances for `read`-like operations.

In order to simplify memory management and to provide optimal memory access safeguards, the SAGA C++ bindings distinguish between `const_buffer` and `mutable_buffer` classes. Both types can be used for `write`-like operations, but only `mutable_buffer` instances can be used for `read`-like operations.

## 6 Using Attributes

### 6.1 Quick Introduction

Attributes in SAGA are handled via the `saga::attribute` interface shown above. That interface allows to set, query, and to inspect specific attributes on those SAGA objects that implement the interface.

### 6.2 Reference

---

```

----- Prototype: saga::attribute -----
namespace saga
{
    namespace attributes
    {
        // common attribute values
        char const * const common_true  = "True";
        char const * const common_false = "False";
    }

    class attribute
    {
    public:
        typedef std::vector<std::string>          strvec_type;
        typedef std::map<std::string, std::string> strmap_type;

        std::string get_attribute      (std::string key) const;
        void         set_attribute      (std::string key,
                                         std::string val);

        strvec_type get_vector_attribute (std::string key) const;
        void         set_vector_attribute (std::string key,
                                         strvec_type val);

        void         remove_attribute    (std::string key);

        strvec_type list_attributes      (void) const;
        strvec_type find_attributes      (std::string pat) const;

        bool         attribute_exists    (std::string key) const;
        bool         attribute_is_readonly (std::string key) const;
        bool         attribute_is_writable (std::string key) const;
        bool         attribute_is_vector  (std::string key) const;
        bool         attribute_is_removable (std::string key) const;
    };
}

```

---

```
}
```

## 6.3 Details

A prominent example is the `saga::job::description` class:

### Code Example

```
saga::job::description jobdef;

std::vector<std::string> args;
args.push_back ("2");

jobdef.set_attribute      ("Executable", "/bin/sleep");
jobdef.set_vector_attribute ("Arguments",  args);

saga::job job = js.create_job (jobdef);
```

That example, as simple as it is, already shows most of what the attribute interface offers—that interface is really simple! For some classes, the set of available attributes is fixed, and will never change. That is also the case for the job description from the example above. For other classes, such as for logical files (`saga::logical_file`), the application programmer can specify arbitrary attributes – their interpretations are, of course, up to the application again.

Note that SAGA defines the known available attributes as static strings. Thus, the above example is more safely written as:

### Code Example

```
saga::job::description jobdef;

std::vector<std::string> args;
args.push_back ("2");

namespace sja = saga::job::attributes;

jobdef.set_attribute      (sja::description_executable, "/bin/sleep");
jobdef.set_vector_attribute(sja::description_arguments, args);

saga::job job = js.create_job (jobdef);
```

Type	Format	Example
String	as in <code>printf ("%s", val);</code>	Hello World
Int	as in <code>printf ("%lld", val);</code>	123
Float	as in <code>printf ("%lld", val);</code>	1.234E-4
Time	as in <code>printf ("%lld", val);</code>	Mon Oct 20 11:31:54 1952
Bool	True or False	True
Enum	literal value of the enum	Done

Table 1: Available types and the prescribed formatting for the attribute value

### 6.3.1 Attribute Types

Although all attributes in SAGA are string-based, we distinguish between different types of attributes. The available types and the prescribed formatting for the attribute values are summarized in Table ??.

Trying to set an attribute which is, for its type, incorrectly formatted, will result in a 'BadParameter' exception.



## 7 Using URLs

### 7.1 Quick Introduction

URLs (and URIs, see below) are a dominant concept for referencing application-external resources. As such, they are also widely used in the Grid world, and in SAGA. The `saga::url` class helps to manage such URLs.

### 7.2 Reference

Prototype: `saga::url`

```
namespace saga
{
    class url
        : public saga::object
    {
    public:
        url (void);
        url (std::string const & urlstr);

        ~url (void);

        url & operator= (std::string const & urlstr);

        std::string get_string    (void) const;

        std::string get_url      (void) const;
        void          set_url    (std::string const & url);

        std::string get_scheme   (void) const;
        void          set_scheme (std::string const & scheme);

        std::string get_host     (void) const;
        void          set_host   (std::string const & host);

        int          get_port    (void) const;
        void          set_port    (int port);

        std::string get_fragment (void) const;
        void          set_fragment (std::string const & fragment);

        std::string get_path     (void) const;
        void          set_path    (std::string const & path);

        std::string get_userinfo (void) const;
        void          set_userinfo (std::string const & userinfo);
    };
}
```

```

        saga::url translate      (std::string const & proto);
};

std::ostream& operator<< (std::ostream      & os,
                          saga::url const & u);
std::istream& operator>> (std::istream      & is,
                          saga::url const & u);
bool          operator== (saga::url const & lhs,
                          saga::url const & rhs);
bool          operator<  (saga::url const & lhs,
                          saga::url const & rhs);
}

```

## 7.3 Details

In the time of the Internet, there are probably only few people in the first and second world who are not aware of the concept of URLs. However, many miss the finer details of the *Uniform Resource Locators*, and for a good reason: URLs are *designed* to hide a number of complexities from the user. In order to make efficient use of the SAGA API, and of many Grid concepts in general, we will need a basic understanding of several key elements of URLs<sup>8</sup>.

### 7.3.1 The Structure of URLs

RFC 3986[?] defines the generic syntax for URIs (and thus for URLs). A URL consists of four parts:

```
<scheme> : <hierarchical part> [ ? <query> ] [ # <fragment> ]
```

The **scheme** defines how the other parts of the URL are interpreted. The remaining parts define what resource the URL points to, and possibly how to access that resource.

The **hierarchical part** usually contains the following information: **userinfo**, **host**, **port**, and **path**. This information specifies **where** the resource is to be found.

The **saga::url** class defines setters and getters for most of these individual

<sup>8</sup>Technically, a URL is a URI that, “in addition to identifying a resource, [provides] a means of locating the resource by describing its primary access mechanism (e.g., its network location).” (“Cool URLs don’t change”, <http://www.w3.org/Provider/Style/URI>) This document, however, uses the terms URL and URI interchangeably.

URL elements. When setting one of these elements, SAGA will make sure that the result is a valid URL—otherwise the setter will decline the operation with a `BadParameter` exception:

Code Example

```
saga::url u ("ftp://remote.host.net:1234/data/old.dat");

u.set_host ("local.host/net");
u.set_path ("/data/new.dat");
```

Line 3 would result in a `BadParameter` exception, as the host contains an invalid character. Line 4 in the example above would change the URL to

```
ftp://remote.host.net:1234/data/new.dat
```

### 7.3.2 URLs in Grids

Now, in Grids we face the problem that different URLs may point to the same resource. For example, the following two URLs may refer to the same file:

```
http://data.silo.net/data/joe_doe/recent/abc.dat
ftp://data.silo.net/pub/joe_doe/recent/abc.dat
```

How can the SAGA API handle these differences transparently? In short: it can't<sup>9</sup>, however, it can help. For example, it allows to translate URLs from one scheme to the other. The code snippet below *may* be able to repeat the printout from above<sup>10</sup>:

Code Example

```
// Translating URLs
saga::url in("http://data.silo.net/data/joe_doe/recent/abc.dat");
saga::url out = in.translate ("ftp://");

std::cout << in.get_string () << std::endl;
std::cout << out.get_string () << std::endl;
```

Also, SAGA allows you to use the special scheme `any://` as a placeholder. If specified, the SAGA implementation tries to guess the correct scheme, and does the potentially required URL translation in the background.

<sup>9</sup>For a detailed discussion see section 2.11 of the SAGA specification [?].

<sup>10</sup>Please note that, at the moment, no adaptor implements `url::translate()`.

**HINT:**

Please note that URL translation is a non-trivial and error prone process, so it is not a good idea to rely on it if you want to keep your application portable (which is a MUST in Grids!). Wherever possible stick to the known schemes, and keep your transactions inside a single scheme and name space.

## Part II

# Using the SAGA-API Packages

As described in the introduction, the SAGA functional packages provide the programmer with the means to interact with Grid resources. Here, we describe these packages one by one, starting with a short overview for each, then a reference section (as C++ declarations of the respective classes and methods), and some details and examples for each of the classes.

## 8 Using the File Package

### 8.1 Quick Introduction

The SAGA filesystem package provides an abstraction to remote file systems. It inherits the SAGA namespace package, so the reader may want to study Section ?? as well.

File systems provide more than just a namespace: they also provide access to the *contents* of files: the `saga::filesystem` package extends the namespace package by adding `read()`, `write()` and `seek()` to the entries (i.e., to `saga::filesystem::file`):

#### Accessing a file

```
// allocate a 'large' buffer statically
char mem[1024];
saga::mutable_buffer buf (mem, 512);

// open a file
saga::url u ("/etc/passwd");
saga::filesystem::file f (u);

// read data into buffer - the first 512 bytes get fill
f.read (buf);

// seek file and buffer
buf.set_data (mem + 512, 512);
f.seek (123, saga::filesystem::Current);

// read again
f.read (buf);

// print what we got
std::cout << mem << std::endl;

return (0);
```

The `saga::buffer` class represents the memory the data are written into. The buffers' memory management can be controlled, but the above example leaves everything to the SAGA implementation—for details, see Section ??.

Note that there are more I/O methods available in the filesystem package. They are, however, mostly provided for optimization, but are really *required* to make remote file I/O operations useful.

Conceptually, the file remains a namespace entry with added read, write and seek capabilities. For more information, see the details below, and also the section "*Namespaces*" (Sec. ??).

## 8.2 Reference

\_\_\_\_\_ saga::filesystem enums \_\_\_\_\_

```
namespace saga
{
    namespace filesystem
    {
        enum flags
        {
            Unknown      = /* -1, */ saga::name_space::Unknown      ,
            None         = /*  0, */ saga::name_space::None         ,
            Overwrite    = /*  1, */ saga::name_space::Overwrite    ,
            Recursive    = /*  2, */ saga::name_space::Recursive    ,
            Dereference  = /*  4, */ saga::name_space::Dereference  ,
            Create       = /*  8, */ saga::name_space::Create       ,
            Exclusive    = /* 16, */ saga::name_space::Exclusive    ,
            Lock         = /* 32, */ saga::name_space::Lock         ,
            CreateParents = /* 64, */ saga::name_space::CreateParents ,
            Truncate     =   128,
            Append       =   256,
            Read         = /* 512, */ saga::name_space::Read         ,
            Write        = /*1024, */ saga::name_space::Write        ,
            ReadWrite    = /*1536, */ saga::name_space::ReadWrite    ,
            Binary       =   2048
        };

        enum seek_mode
        {
            Start  = 1,
            Current = 2,
            End    = 3
        };
    }
}
```

\_\_\_\_\_ Prototype: saga::filesystem::iovec \_\_\_\_\_

```
namespace saga
{
    namespace filesystem
    {
        class const_iovec
        {
            : public saga::const_buffer
        {

```

```

    public:
        const_iovec (void const * data,
                     saga::ssize_t size,
                     saga::ssize_t len_in = -1);
        ~const_iovec (void);

        saga::ssize_t get_len_in (void) const;
        saga::ssize_t get_len_out (void) const;

};

class iovec
    : public saga::mutable_buffer
{
    public:
        iovec (void * data = 0,
               saga::ssize_t size = -1,
               saga::ssize_t len_in = -1,
               buffer_deleter cb = default_buffer_deleter);
        ~iovec (void);

        void set_len_in (saga::ssize_t len_in);
        saga::ssize_t get_len_in (void) const;
        saga::ssize_t get_len_out (void) const;

};
}
}

```

Prototype: `saga::filesystem::file`

```

namespace saga
{
    namespace filesystem
    {
        class file
            : public saga::name_space::entry
        {
            public:
                file (saga::session const & s,
                     saga::url url,
                     int mode = Read);
                file (saga::url url,
                     int mode = Read);
                file (saga::object const & o);
                file (void);
                ~file (void);

                file & operator= (saga::object const & o);
        };
    };
}

```



```

    saga::off_t    get_size (void);

    saga::ssize_t read      (saga::mutable_buffer buffer,
                             saga::ssize_t        length = 0);
    saga::ssize_t write     (saga::const_buffer  buffer,
                             saga::ssize_t        length = 0);
    saga::off_t    seek     (saga::off_t        offset,
                             seek_mode          mode);

    void           read_v   (std::vector<iovec>  buffer_vec);
    void           write_v  (std::vector<const_iovec>
                             buffer_vec);

    saga::ssize_t size_p    (std::string        pattern);
    saga::ssize_t read_p    (std::string        pattern,
                             saga::mutable_buffer buffer);
    saga::ssize_t write_p   (std::string        pattern,
                             saga::const_buffer  buffer);

    std::vector<std::string>
    modes_e (void);
    saga::size_t size_e     (std::string        ext_mode,
                             std::string        specification);
    saga::ssize_t read_e    (std::string        ext_mode,
                             std::string        specification,
                             saga::mutable_buffer buffer);
    saga::ssize_t write_e   (std::string        ext_mode,
                             std::string        specification,
                             saga::const_buffer  buffer);

};
}
}

```

Prototype: `saga::filesystem::directory`

```

namespace saga
{
    namespace filesystem
    {
        class directory
        : public saga::name_space::directory
        {
        public:
            directory (saga::session const & s,
                      saga::url          url,
                      int                 mode = ReadWrite);
            directory (saga::url          url,
                      int                 mode = ReadWrite);
            directory (saga::object const & o);
        };
    };
}

```

```
    directory (void);
    ~directory (void);

    directory & operator= (saga::object const & o);

    saga::off_t get_size (saga::url url);
    bool        is_file  (saga::url url);
    file        open     (saga::url url,
                          int      flags = Read);
    directory   open_dir (saga::url url,
                          int      flags = ReadWrite);
}
}
```

### 8.3 Filesystem Details

The described classes are syntactically and semantically POSIX-oriented [?, ?, ?]. Executing large numbers of simple POSIX-like remote data access operations is, however, prone to latency-related performance problems. To allow for efficient implementations, the presented API borrows ideas from GridFTP and other specifications which are widely used for remote data access. These extensions should be seen as just that: optimizations. Be aware that the SAGA adaptors usually implement the POSIX-like `read()`, `write()` and `seek()` methods, and rarely implement the additional optimized methods (a 'NotImplemented' exception is thrown if these are not implemented). The optimizations included here are:

**Scattered I/O** Scattered I/O operations are already defined by POSIX, as `readv()` and `writenv()`. Essentially, these methods represent vector versions of the standard POSIX `read()/write()` methods; the arguments are, basically, vectors of instructions to execute, and buffers to operate upon. In other words, `readv()` and `writenv()` can be regarded as specialized bulk methods, which cluster multiple I/O operations into a single operation. The advantages of such an approach are that it is easy to implement, it is very close to the original POSIX I/O in semantics, and, in some cases, it is even very fast. The disadvantage is that for many small I/O operations (a common occurrence in SAGA use cases), the description of the I/O operations can be larger than the sent, returned or received data.

**Pattern-Based I/O (FALLS)** One approach to address the bandwidth limitation of scattered I/O is to describe the required I/O operations at a more

abstract level. Regularly, repeating patterns of binary data can be described by the so-called 'FamiLy of Line Segments' (FALLS) [?]. The pattern-based I/O routines in SAGA use such descriptions to reduce the bandwidth limitation of scattered I/O. The advantage of such an approach is that it targets very common data access patterns (at least those commonly found in SAGA use cases). The disadvantages are that FALLS is a paradigm not widely known or used, and that FALLS is by definition, limited to regular patterns of data, hence the inefficiency for more randomized data access.

FALLS was originally introduced for transformations in parallel computing. There is also a parallel filesystem which uses FALLS to describe the file layout. FALLS can be used to describe regular subsets of arrays with a very compact syntax.

FALLS patterns are formed as 5-tuples: "(from,to,stride,rep,(pat))". The **from** element defines the starting offset for the first pattern unit; **to** defines the finishing offset of the first pattern unit; **stride** defines the distance between consecutive pattern units (start to start); and **rep** defines the number of repetitions of the pattern units. The optional 5th element, **pat**, allows to define nested patterns, where the internal pattern defines the unit the outer pattern is applied to (by default it is one byte). As an example, the following FALLS describe the highlighted elements of the matrix in Fig. ??: "(0,17,36,6,(0,0,2,6))": the inner pattern describes a pattern unit of one byte length (from 0 to 0), with a distance of 2 to the next application, and 6 repetitions. These are the 6 bytes per line which are marked. The outer pattern defines the repeated application of the inner pattern, starting at 0, ending at 17 (end of line), distance of 36 (to the beginning of next but one line), and repetition of 6.

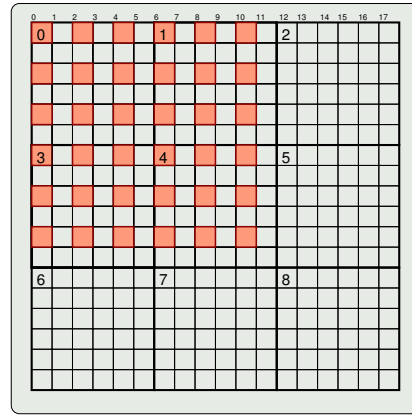


Figure 1: The highlighted elements are defined by "(0,17,36,6,(0,0,2,6))".

**Extended I/O** GridFTP (which was designed for a similar target domain) introduced an additional remote I/O paradigm, that of Extended I/O operations.

In essence, the Extended I/O paradigm allows the formulation of I/O requests using custom strings, which are not interpreted on the client, but on the server side; these can be expanded to arbitrarily complex sets of I/O operations. The type of I/O request encoded in the string is called **mode**. A server may support one or many of these extended I/O modes. Whereas the approach is very flexible

and powerful and has proven its usability in GridFTP, a disadvantage is that it requires very specific infrastructure to function, i.e., it requires a remote server instance which can interpret opaque client requests. Additionally, no client side checks or optimizations on the I/O requests are possible. Also, the application programmer needs to estimate the size of the data to be returned in advance, which in some cases is very difficult.

The three described operations have, if compared to each other, increasing semantic flexibility, and are increasingly powerful for specific use cases. However, they are also increasingly difficult to implement and support in a generic fashion. It is up to the SAGA adaptors and the specific use cases, to determine the level of I/O abstraction that serves the application best and that can be best supported in the target environment.

### **Enum flags**

The enum **flags** are inherited from the **namespace** package. A number of file specific flags are added to it. All added flags are used for the opening of **file** and **directory** instances, and are not applicable to the operations inherited from the **namespace** package.

#### **Truncate**

Upon opening, the file is truncated to length 0, i.e., a following **read()** operation will never find any data in the file. That flag does not apply to directories.

#### **Append**

Upon opening, the file pointer is set to the end of the file, i.e., a following **write()** operation will extend the size of the file. That flag does not apply to directories.

#### **Read**

The file or directory is opened for reading—that does not imply the ability to write to the file or directory.

#### **Write**

The file or directory is opened for writing—that does not imply the ability to read from the file or directory.

#### **ReadWrite**

The file or directory is opened for reading and writing.

#### **Binary**

Some operating systems (notably windows-based systems) distinguish between binary and non-binary modes—this flag mimics that behaviour.

### Class `iovec`

The `iovec` class inherits the `saga::buffer` class, and three additional state attributes: `offset`, `len_in` and `len_out` (with the latter one being read-only). With that addition, the new class can be used very much the same way as the `iovec` structure defined by POSIX for `readv/writev`; the buffer `len_in` is being interpreted as the POSIX `iov_len`, i.e., the number of bytes to read/write.

If `len_in` is not specified, that length is set to the size of the buffer. For application-managed buffers, it is a `BadParameter` error if `len_in` is specified to be larger than size, (see Section ?? for details on buffer memory management). Before an `iovec` instance is used, its `len_in` must be set to a non-zero value; otherwise its use will cause a `BadParameter` exception.

After a `read_v()` or `write_v()` operation completes, `len_out` will report the number of bytes read or written. Before completion, the SAGA implementation will report `len_out` to be `-1`.

## 9 Using the Replica Package

### 9.1 Quick Introduction

Another package inheriting the namespace package is the replica management in `saga::replica`. It introduces logical files, which are namespace entries that have a number of locations (URLs) attached pointing to identical physical copies of the same file. The replica package is again a very simple extension of the namespace package: the class `logical_file` is a namespace `entry` class with a number of additional methods which allow to manage the list of attached URLs:

#### Managing replica locations

```
// open a logical file
saga::url u ("/replica_1");
saga::replica::logical_file lf (u, saga::replica::Create
                               | saga::replica::ReadWrite);

// Add a replica location, replicate the file
lf.add_location (saga::url ("file://localhost/etc/passwd"));
lf.replicate    (saga::url ("file://localhost/tmp/passwd"),
                saga::replica::Overwrite);

// list all locations
std::vector <saga::url> replicas = lf.list_locations ();

for ( unsigned int i = 0; i < replicas.size (); i++ )
{
    std::cout << "replica: " << replicas[i] << std::endl;
}

// remove the first location
lf.remove_location (replicas[0]);
```

Managing replicas is fairly simple, because it only covers methods to list, add, delete, and update replica locations on logical files. Additionally, logical files and logical directories can have meta data attached, which are sets of string-based key-value pairs<sup>11</sup>:

#### Managing replica meta data

```
// open a logical file
saga::url u ("/replica_1");
saga::replica::logical_file lf (u, saga::replica::Create
                               | saga::replica::ReadWrite);
```

<sup>11</sup>For brevity, the example does not distinguish between scalar and vector attributes.

```
// get all attributes
std::vector<std::string> keys = lf.list_attributes ();

// print the keys and values
for ( unsigned int i = 0; i < keys.size (); i++ )
{
    std::string key = keys[i];
    std::string val;

    if ( lf.attribute_is_vector (key) )
    {
        std::vector<std::string> vals = lf.get_vector_attribute (key);
        val = vals[0] + " ...";
    }
    else
    {
        val = lf.get_attribute (key);
    }
    std::cout << key << " -> " << val << std::endl;
}
```

The attentive reader will notice that the attribute management is in accordance with the attribute management part of the SAGA Look & Feel.

## 9.2 Reference

```
----- Prototypes: saga::replica -----

namespace saga
{
    namespace replica
    {
        namespace metrics
        {
            char const * const logical_file_modified
                = "logical_file.Modified";
            char const * const logical_file_deleted
                = "logical_file.Deleted";
            char const * const logical_directory_created_entry
                = "logical_directory.CreatedEntry";
            char const * const logical_directory_modified_entry
                = "logical_directory.ModifiedEntry";
            char const * const logical_directory_deleted_entry
                = "logical_directory.DeletedEntry";
        }

        enum flags
```

```

{
    Unknown      = /* -1, */  saga::name_space::Unknown      ,
    None         = /*  0, */  saga::name_space::None         ,
    Overwrite    = /*  1, */  saga::name_space::Overwrite    ,
    Recursive    = /*  2, */  saga::name_space::Recursive    ,
    Dereference  = /*  4, */  saga::name_space::Dereference  ,
    Create       = /*  8, */  saga::name_space::Create       ,
    Exclusive    = /* 16, */  saga::name_space::Exclusive    ,
    Lock         = /* 32, */  saga::name_space::Lock         ,
    CreateParents = /* 64, */  saga::name_space::CreateParents ,
                                // 128,      reserved for Truncate
                                // 256,      reserved for Append
    Read         =   512,
    Write        =  1024,
    ReadWrite    =  1036,
                                // 2048,      reserved for Binary
};

class logical_file
: public saga::name_space::entry,
  public saga::attributes
{
public:
    logical_file (session const      & s,
                  saga::url          url,
                  int                 mode = Read);
    logical_file (saga::url          url,
                  int                 mode = Read);
    logical_file (saga::object const & o);
    logical_file (void);
    ~logical_file (void);

    logical_file & operator= (saga::object const & o);

    void add_location   (saga::url url);
    void remove_location (saga::url url);
    void update_location (saga::url old,
                          saga::url new);
    std::vector<saga::url>
        list_locations (void);
    void replicate      (saga::url url,
                          int      flags = None);
};

class logical_directory
: public saga::name_space::directory,
  public saga::attributes
{
public:
    logical_directory (saga::session const & s,

```



```
        saga::url          url,
        int                mode = ReadWrite);
logical_directory (saga::url          url,
                  int                mode = ReadWrite);
logical_directory (saga::object const & o);
logical_directory (void);
~logical_directory (void);

logical_directory & operator=(saga::object const& o);

bool is_file      (saga::url url);
std::vector<saga::url>
    find          (std::string          name_pattern,
                  std::vector<std::string> key_pattern,
                  int                flags = Recursive);

saga::replica::logical_file
    open          (saga::url          url,
                  int                flags = Read);
saga::replica::logical_directory
    open_dir      (saga::url          url,
                  int                flags = None);
};
}
}
```

### 9.3 Replica Details

## 10 Using the Namespace Package

### 10.1 Quick Introduction

Namespaces are used for a wide variety of computer subsystems: they are used to organize files, to manage domain names, web content, etc. Hierarchical namespaces are prevalent for managing files (indeed, most file systems provide a hierarchical namespace, the directory tree). SAGA provides the `saga::name_space` package to navigate and manipulate such namespaces:

————— Navigating a name space —————

```
#include <saga/saga.hpp>

int main (int argc, char** argv)
{
    // open a namespace directory
    saga::url u (std::string ("file://localhost/") + getenv ("PWD"));
    saga::name_space::directory d (u);

    // list the contents
    std::vector <saga::url> entries = d.list ();

    // print the entries, and their type
    for ( unsigned int i = 0; i < entries.size (); i++ )
    {
        std::string type;
        // get some details for the entry
        if ( d.is_dir (entries[i]) )
        {
            type = "/";
        }
        // if a link (symbolic)
        else if ( d.is_link (entries[i]) )
        {
            type = " -> " + d.read_link (entries[i]).get_string();
        }
        // print the info
        std::cout << entries[i] << type << std::endl;
    }

    return (0);
}
```

This is a poor man's `ls`! It needs only five SAGA calls, which are all very simple. Note that a directory has a 'cwd', a Current Working Directory. Calling `cd()`

on a directory changes that. Relative file names are always interpreted with respect to that cwd.

So, the `saga::name_space` package provides two classes: `directory` and `entry` (note that `directory` inherits `entry`). On entries, you can perform `copy()`, `link()`, `move()`, and `remove()`. You can also inspect entries, with `is_dir()`, `is_entry()`, `is_link()` and `read_link()`.

Directories provide the same operations, but with an additional `source` argument, as shown below:

```
entry.copy (target);           // copy entry      to target
dir.copy (target);             // copy dir        to target
dir.copy (source, target);     // copy dir/source to target
```

But the `directory` copy should be recursive, of course:

```
dir.copy (target, saga::name_space::Recursive);
```

Additionally, the `directory` has two open methods:

```
saga::entry    e = dir.open    (saga::url (entry_name));
saga::directory d = dir.open_dir (saga::url (dir_name) );
```

## 10.2 Reference

Prototype: `saga::namespace::flags`

```
namespace saga
{
    namespace name_space
    {
        enum flags
        {
            Unknown      = -1,
            None         = 0,
            Overwrite     = 1,
            Recursive     = 2,
            Dereference   = 4,
            Create        = 8,
            Exclusive     = 16,
            Lock          = 32,
            CreateParents = 64
        };
    }
}
```

Prototype: `saga::namespace::entry`

```
namespace saga
{
    namespace name_space
    {
        class entry
        : public saga::object,
          public saga::monitorable,
          public saga::permissions
        {
        entry (session const & s,
              saga::url      url,
              int             mode = None);
        entry (saga::url      url,
              int             mode = None);
        entry (void);
        ~entry (void);

        // inspection methods
        saga::url get_url  (void) const;
        saga::url get_cwd  (void) const;
        saga::url get_name (void) const;
        saga::url read_link (void) const;

        bool      is_dir   (void) const;
        bool      is_entry (void) const;
        bool      is_link  (void) const;

        // management methods
        void      copy      (saga::url target,
                             int flags = saga::name_space::None);
        void      link      (saga::url target,
                             int flags = saga::name_space::None);
        void      move      (saga::url target,
                             int flags = saga::name_space::None);
        void      remove    (int flags = saga::name_space::None);
        void      close     (double timeout = 0.0);
        };
    }
}
```

Prototype: `saga::namespace::directory`

```
namespace saga
{
    namespace name_space
    {
        class directory
```

```

    : public saga::name_space::entry
{
public:
    directory (session const & s,
               saga::url      url,
               int             mode = None);
    directory (saga::url      url,
               int             mode = None);
    directory (void);
    ~directory (void);

    void      change_dir (saga::url      target)
    std::vector<saga::url>
        list      (std::string pattern = "*",
                   int      flags   = None) const;
    std::vector<saga::url>
        find      (std::string pattern,
                   int      flags   = Recursive) const;

    saga::url  read_link (saga::url      url) const;
    bool       exists    (saga::url      url) const;
    bool       is_dir    (saga::url      url) const;
    bool       is_entry  (saga::url      url) const;
    bool       is_link   (saga::url      url) const;
    unsigned   int get_num_entries
        (void) const;
    saga::url  get_entry  (unsigned int entry) const;
    void       copy      (saga::url      source_url,
                           saga::url      dest_url,
                           int             flags = None);
    void       link      (saga::url      source_url,
                           saga::url      dest_url,
                           int             flags = None);
    void       move      (saga::url      source_url,
                           saga::url      dest_url,
                           int             flags = None);
    void       remove    (saga::url      url,
                           int             flags = None);
    void       make_dir  (saga::url      url,
                           int             flags = None);
    entry      open      (saga::url      url,
                           int             flags = None);
    directory  open_dir  (saga::url      url,
                           int             flags = None);

};
} // namespace_dir
} // namespace saga

```

## 10.3 Details

The notion of a namespace is shared by several SAGA packages: the file, the replica and the advert packages. All allow to manage entities which are organized in a hierarchical namespace. The namespace package is abstracting the management of that hierarchy, and leaves only the entity specific operations (File-IO, Replica Management, Advert Creation) to the various packages.

**HINT:**

Namespace operations handle namespace entries as opaque, and are never able to look 'inside'. To do that, use one of the derived packages, which add exactly that functionality, specific to the type of namespace they represent.

The namespace package introduces two classes: `saga::namespace::entry` and `saga::namespace::directory` (which is an entry, i.e., inherits from `entry`). Both classes refer to entities which are specified by a pathname, typically a URL. Several calls additionally allow to refer to entries with wildcards, similar to the shell wildcards known by POSIX (see `glob(7)`)<sup>12</sup>.

The namespace package allows to create and delete entries, to copy, move or link entries, and to inspect entries and directories.

---

<sup>12</sup>Namespace entries can also have permissions, just as in most file systems. For details on permissions, see Section ??.

## 11 Using the Job Package

### 11.1 Quick Introduction

For an overwhelming number of use cases, job submission and management is the most important aspect of Grid computing. The `saga::job` package provides the respective functionality. There are four classes:

- `saga::job::description`  
describes the properties of a job to be submitted,
- `saga::job::service`  
represents a service which accepts job descriptions for submission,
- `saga::job::job`  
represents the submitted job, and
- `saga::job::self`  
which represents the application itself.

The most basic code example is the following:

#### Job submission

```
#include <saga/saga.hpp>

int main (int argc, char** argv)
{
    saga::job::service js;
    saga::job::job j = js.run_job ("localhost", "/bin/sleep 3");
    j.wait ();

    return (0);
}
```

Yes, it is that easy. But what about the job's I/O, can we handle that as well? And what about job state information <F2>? Let's see:

#### Job submission and job I/O

```
saga::job::service js;
saga::job::ostream in;
saga::job::istream out;
saga::job::istream err;

saga::job::job j = js.run_job ("localhost",
                               "/bin/date",
```

```
                                in, out, err);

saga::job::state state = j.get_state ();

do
{
    char buffer[256];

    // get stdin
    out.read (buffer, sizeof (buffer));
    if ( out.gcount () > 0 )
    {
        std::cout << std::string (buffer, out.gcount ());
    }

    if ( out.fail () )
    {
        break;
    }

    usleep (10000);
    state = j.get_state ();
} while ( state != saga::job::Done &&
          state != saga::job::Failed &&
          state != saga::job::Canceled )
```

That is basically the same as above, but we catch the job's I/O channel as it is getting created. The remainder of the example code (lines 12ff) is just watching the job state and the output stream.

The `run_job()` method is just a shortcut. It actually provides exactly the following functionality, just with less code, as the example below:

```
_____ run_job() expanded _____

namespace sa = saga::attributes;
namespace sja = saga::job::attributes;

std::string exe ("/bin/date");

std::vector <std::string> hosts;
hosts.push_back ("localhost");

saga::job::description jd;

jd.set_attribute      (sja::description_interactive, sa::common_true);
jd.set_attribute      (sja::description_executable, exe);
```



```
jd.set_vector_attribute (sja::description_candidatehosts, hosts);

saga::job::service js;

saga::job::job j = js.create_job (jd);

saga::job::ostream in = j.get_stdin ();
saga::job::istream out = j.get_stdout ();
saga::job::istream err = j.get_stderr ();

// job is in 'New' state here, we need to run it
j.run ();

saga::job::state state = j.get_state ();

do
{
    char buffer[256];

    // get stdin
    out.read (buffer, sizeof (buffer));
    if ( out.gcount () > 0 )
    {
        std::cout << std::string (buffer, out.gcount ());
    }
    if ( out.fail () )
    {
        break;
    }
    usleep (10000);
    state = j.get_state ();
} while ( state != saga::job::Done &&
          state != saga::job::Failed &&
          state != saga::job::Canceled )
```

This long way exposes some additional details:

- The `job_description` class allows specifying attributes of the job.<sup>13</sup>
- The `stdio` channels of a job can be obtained before the job is running, to avoid race conditions.
- Only 'Interactive' jobs will provide `stdio` channels.

---

<sup>13</sup>Only the 'Executable' attribute is mandatory.

Table ?? shows the job description attributes available in SAGA. Note that these attributes are closely related to the respective keys of the JSDL standard [?, ?].

Executable	string	what executable to run
Arguments	list of strings	the job arguments
CandidateHosts	list of strings	where to run
SPMDVariation	string	MPI type (if any)
TotalCPUCount	int	# CPUs, total
NumberOfProcesses	int	# processes, total
ProcessesPerHost	int	# processes per host
ThreadsPerProcess	int	# threads per process
Environment	list of strings	environment vars
WorkingDirectory	string	working directory
Interactive	bool	usage type
Input	string	stdin source file
Output	string	stdout log file
Error	string	stderr log file
FileTransfer	list of strings	file staging
Cleanup	bool	cleanup-after-run
JobStartTime	datetime	when to start job
TotalCPUTime	int	how long will job run
TotalPhysicalMemory	int	memory the job needs
CPUArchitecture	string	architecture to run on
OperatingSystemType	string	OS to run on
Queue	string	queue to submit to
JobContact	list of strings	notification contact

Table 2: Job description attributes available in SAGA.

The available `SPMDVariation` values are: `MPI`, `GridMPI`, `IntelMPI` `LAM-MPI`, `MPICH1`, `MPICH2`, `MPICH-GM`, `MPICH-MX` `MVAPICH`, `MVAPICH2`, `OpenMP`, `POE`, `PVM` and `None`. Other arbitrary values are allowed, and interpreted by the backend. This attribute indirectly determines which `mpirun` executable should be called for starting the application, and which parameters it will accept.

Line 10 in the job I/O example above obtains the job state:

```
saga::job::state state = j.get_state ();
```

That implies that `saga::job` is a stateful object. The job state model is shown in Figure ??, which also shows what method calls cause which state transitions.

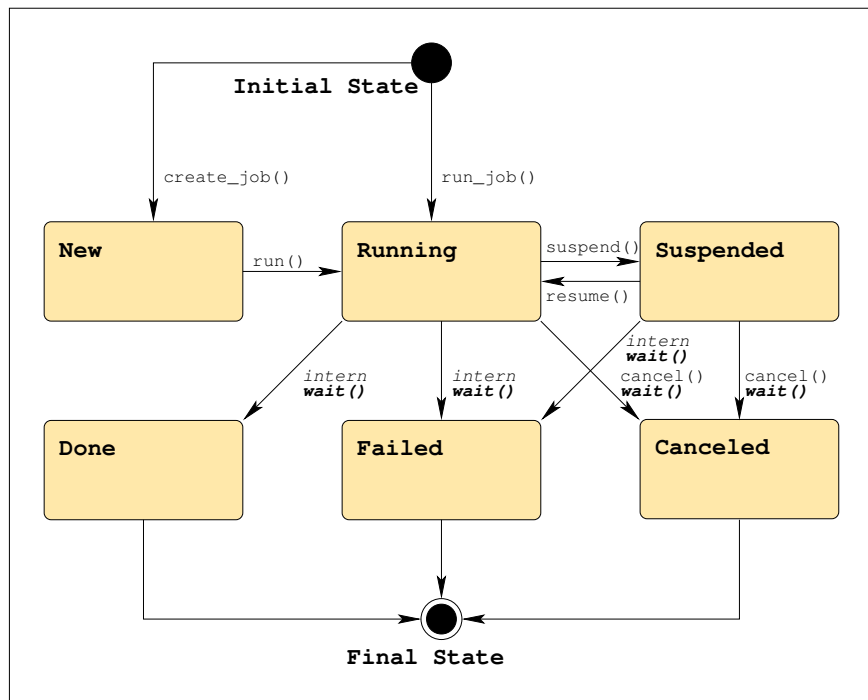


Figure 2: The SAGA job state model.

## 11.2 Reference

Prototypes: `saga::job`

```

namespace saga
{
    namespace job
    {
        typedef char const * char ccc_type;

        namespace attributes
        {
            // job description attributes
            ccc_type description_executable           = "Executable";
            ccc_type description_arguments            = "Arguments";
            ccc_type description_environment          = "Environment";
            ccc_type description_workingdirectory    = "WorkingDirectory";
            ccc_type description_interactive         = "Interactive";
            ccc_type description_input               = "Input";
            ccc_type description_output              = "Output";
            ccc_type description_error               = "Error";
            ccc_type description_filetransfer        = "FileTransfer";
        }
    }
}

```

```

ccc_type description_cleanup          = "Cleanup";
ccc_type description_jobstarttime     = "JobStartTime";
ccc_type description_totalcputime     = "TotalCPUTime";
ccc_type description_totalphysicalmemory = "TotalPhysicalMemory";
ccc_type description_cpuarchitecture = "CPUArchitecture";

ccc_type cpuarchitecture_sparc       = "sparc";
ccc_type cpuarchitecture_powerpc     = "powerpc";
ccc_type cpuarchitecture_x86         = "x86";
ccc_type cpuarchitecture_x86_32      = "x86_32";
ccc_type cpuarchitecture_x86_64      = "x86_64";
ccc_type cpuarchitecture_parisc      = "parisc";
ccc_type cpuarchitecture_mips        = "mips";
ccc_type cpuarchitecture_ia64        = "ia64";
ccc_type cpuarchitecture_arm         = "arm";
ccc_type cpuarchitecture_other       = "other";

ccc_type description_operatingsystemtype = "OperatingSystemType";
ccc_type description_candidatehosts     = "CandidateHosts";
ccc_type description_queue              = "Queue";
ccc_type description_jobcontact         = "JobContact";
ccc_type description_spmddvariation     = "SPMDVariation";
ccc_type description_totalcpucount      = "TotalCPUCount";
ccc_type description_numberofprocesses  = "NumberOfProcesses";
ccc_type description_processesperhost   = "ProcessesPerHost";
ccc_type description_threadspersprocess = "ThreadsPerProcess";
}

class description
: public saga::object,
  public saga::attributes
{
public:
    description (void);
    ~description (void);
};

class service
: public saga::object
{
public:
    service (saga::session const & s,
            saga::url          rm = "");
    service (saga::url          rm = "");
    ~service (void);

    saga::job::job create_job (description job_desc);

    saga::job::job run_job    (std::string  hostname,

```

```

                                std::string  cmdline,
                                ostream      & stdin_stream,
                                istream      & stdout_stream,
                                istream      & stderr_stream);

    std::vector<std::string>
        list      (void);
    saga::job::job get_job  (std::string  job_id);
    saga::job::self get_self (void);
};

namespace attributes
{
    // read only job attributes
    ccc_type jobid          = "JobID";
    ccc_type executionhosts = "ExecutionHosts";
    ccc_type created        = "Created";
    ccc_type started        = "Started";
    ccc_type finished       = "Finished";
    ccc_type workingdirectory = "WorkingDirectory";
    ccc_type exitcode        = "ExitCode";
    ccc_type termsig         = "Termsig";
}

namespace metrics
{
    // job metrics
    ccc_type statedetail    = "job.StateDetail";
    ccc_type signal          = "job.Signal";
    ccc_type cputimelimit   = "job.CPUTimeLimit";
    ccc_type memoryuse       = "job.MemoryUse";
    ccc_type vmemoryuse     = "job.VmemoryUse";
    ccc_type performance    = "job.Performance";
}

enum state
{
    // job state
    Unknown = saga::task_base::Unknown, // -1
    New     = saga::task_base::New,      // 0
    Running = saga::task_base::Running,  // 1
    Failed  = saga::task_base::Failed,   // 2
    Done    = saga::task_base::Done,     // 3
    Canceled = saga::task_base::Canceled, // 4
    Suspended = 5
};

class job : public saga::task,
            public saga::attributes,
            public saga::permissions,

```

```

        public saga::sync
    {
        public:
            // no constructor
            ~job (void);

            job & operator= (saga::object const & o);

            std::string get_job_id      (void);
            state      get_state      (void);
            description get_description (void);
            ostream    get_stdin      (void);
            istream    get_stdout     (void);
            istream    get_stderr     (void);

            void        suspend        (void);
            void        resume         (void);

            void        checkpoint     (void);
            void        migrate        (description job_desc);
            void        signal         (int         signal);
    };
}

```

### 11.3 Job Details

The SAGA Job API covers four classes: `saga::job_description`, `saga::job_service`, `saga::job`, and `saga::job_self`. The job description class is nothing more than a container for a well-defined set of attributes, which, using JSDL [?, ?] based keys, defines the job to be started, and its runtime and resource requirements. The job server represents a resource management endpoint which allows the starting and insertion of jobs.

The job class itself is central to the API, and represents an application instance running under the management of a resource manager. The `job_self` class IS-A job. Its purpose is to represent the current SAGA application, which allows for a number of use cases with applications that actively interact with the grid infrastructure, for example, to migrate itself, or to set new job attributes.

The job class inherits the `saga::task` class (Sec. ??), and uses its methods to `run()`, to `wait()` for, and to `cancel()` jobs. The inheritance also allows for the management of large numbers of jobs in task containers. Additional methods provided by the `saga::job` class relate to the `Suspended` state (which is not available on tasks), and provide access to the job's standard I/O streams, and

to more detailed status information. The `saga::job` iostreams are available as `saga::job::istream` and `saga::job::ostream` instances, which all inherit from `std::stream`.

### 11.3.1 Job State Model

The SAGA job state diagram is shown in Figure ?? . It is an extension of the `saga::task` state diagram (Figure ??), and extends the state diagram with a 'Suspended' state, which the job can enter/leave using the `suspend()`/`resume()` calls.

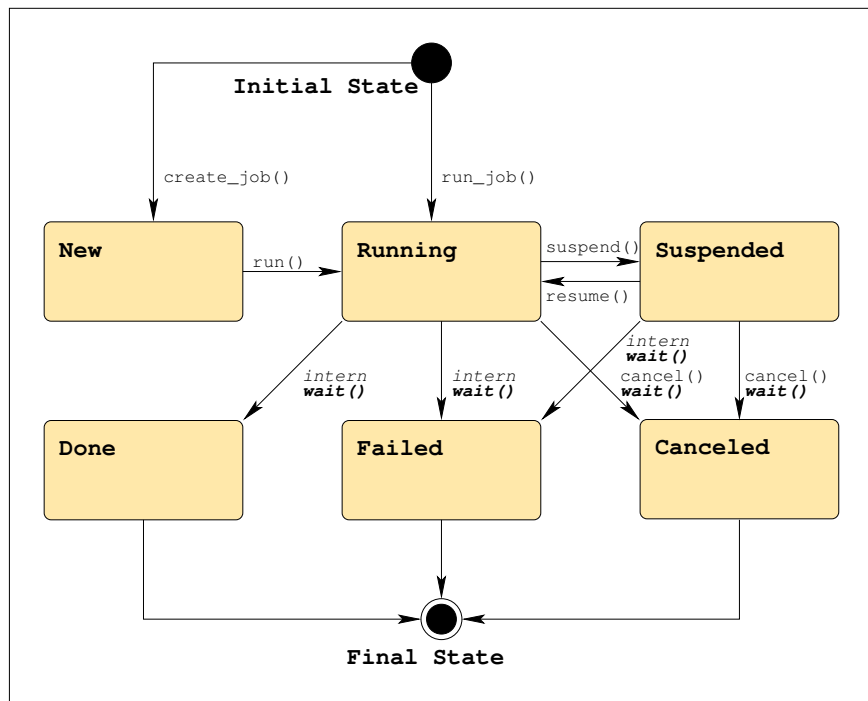


Figure 3: The SAGA job state model extends the SAGA task state model with a 'Suspended' state, and additional transitions.

SAGA maps the native back-end state model onto the SAGA state model. The SAGA state model is simple enough to allow a straightforward mapping in most cases. For some applications, access to the native back-end state model is useful, though—for that reason, an additional metric named '`StateDetail`' allows to query the native job state.

State details in SAGA are formatted as follows:

`'<model>:<state>'`

with valid models being "BES", "DRMAA", or other implementation specific models. For example, a state detail for the BES state 'StagingIn' would be rendered as 'BES:StagingIn', and would be a substate of **Running**. If no state details are available, the metric is still available, but it has always an empty string value.

### 11.3.2 File Transfer Specifications

The syntax of a file transfer directive for the job description is modeled on the LSF syntax (LSF stands for *Load Sharing Facility*, a commercial job scheduler by Platform Computing). It has the general syntax:

`local_file operator remote_file`

Both the `local_file` and the `remote_file` can be URLs. If they are not URLs, but full or relative pathnames, then the `local_file` is relative to the host where the submission is executed, and the `remote_file` is evaluated on the execution host of the job.

The operator is one of the following four:

- `'>'` copies the local file to the remote file before the job starts.  
Overwrites the remote file if it exists.
- `'>>'` copies the local file to the remote file before the job starts.  
Appends to the remote file if it exists.
- `'<'` copies the remote file to the local file after the job finishes.  
Overwrites the local file if it exists.
- `'<<'` copies the remote file to the local file after the job finishes.  
Appends to the local file if it exists.

### 11.3.3 Command Line Specification

The `run_job()` method of the `saga::job_service` class accepts a string parameter which constitutes a command line to be executed on a remote resource. The parsing of that command line follows the following rules:

- Elements are either delimited by white space (a space, or a tab), or are delimited by not-escaped double quotes.
- The escape character for double quotes is the backslash (which is also used to escape the backslash itself).



- The first element is used as the executable name; all other elements are treated as job arguments.

The following command line parameter

```
/bin/cp "/tmp/my file with spaces" /data/
```

is thus invoking the executable `/bin/cp`, with two arguments.

#### 11.3.4 Job Identifiers

The SAGA `JobID` is treated as an opaque string in the SAGA API, and is structured as:

```
'[backend url]-[native id]'
```

For example, a job submitted to the host `remote.host.net` via `ssh` (whose daemon runs on port 22), and having the POSIX PID 1234, would get the job id:

```
'[ssh://remote.host.net:22/]-[1234]'
```

The implementation may free the resources used for the job, and hence may invalidate a `JobID` after a successful wait on the job, and after the application received the job status information at least once.

A `JobID` may be unknown until the job enters the `Running` state, as the backend will often not assign IDs to jobs which are not yet running. In such cases, the value of the `JobID` attribute is empty. The job will, however, retain its `JobID` after it enters a final state.

## 12 Using the Stream Package

### 12.1 Quick Introduction

**(FIXME: english)**

Inter-process communication (IPC) is historically central to parallel and distributed computing, and numerous approaches exist for the wide variety of use cases. Streams are amongst the most well-known and most simple of these paradigms, and are also provided by SAGA. The concept is very close to that of BSD sockets: a `stream_service` can be listened to (calling `serve()`), and a connection client `stream` results in a `stream` instances on both ends, upon which the application can call `read()` and `write()`:

Stream server

```
#include <saga/saga.hpp>

int main (int argc, char** argv)
{
    // reusable io buffer
    saga::buffer buf;

    // create a stream_server and listen for clients
    saga::stream::server ss (saga::url (argv[1]));

    while ( 1 )
    {
        // wait for incoming client
        saga::stream::client s = ss.serve ();

        // read data from client
        s.read (buf);
    }
}
```

Stream client

```
#include <saga/saga.hpp>

int main (int argc, char** argv)
{
    // create client stream and connect to server
    saga::stream::client s (saga::url (argv[1]));

    s.connect ();

    // send some data
```

```
s.write (saga::buffer ("Hello"));
}
```

Note that SAGA is silent about the used wire protocol. In particular, server and client are responsible for picking a compatible transfer mechanism. Also, the bootstrapping mechanism is out of scope in the SAGA stream package, i.e., both sides need to agree on the URL for the server endpoint out-of-band. Both issues will be addressed in future SAGA API extensions (message API and advert API extension).

In particular, for stream-based communication, asynchronous operations and notifications are extremely useful programming paradigms. Both are provided within SAGA, more details can be found in the later description of the SAGA Look & Feel (`saga::monitoring` and `saga::task` package).

## 12.2 Reference

```
----- Prototypes: saga::stream -----
namespace saga
{
    namespace stream
    {
        namespace attributes
        {
            char const * const stream_bufsize      = "Bufsize";
            char const * const stream_timeout      = "Timeout";
            char const * const stream_blocking     = "Blocking";
            char const * const stream_compression  = "Compression";
            char const * const stream_nodelay      = "Nodelay";
            char const * const stream_reliable     = "Reliable";
        }

        namespace metrics
        {
            char const * const stream_state        = "stream.State";
            char const * const stream_read         = "stream.Read";
            char const * const stream_write        = "stream.Write";
            char const * const stream_exception    = "stream.Exception";
            char const * const stream_dropped      = "stream.Dropped";

            char const * const server_clientconnect = "server.ClientConnect";
        }

        enum state
        {
            Unknown      = -1,
        }
    }
}
```

```
New          = 1,
Open         = 2,
Closed       = 3,
Dropped      = 4,
Error        = 5
};

enum activity
{
    Read       = 1,
    Write      = 2,
    Exception   = 4
};

class stream
    : public saga::object,
      public saga::attributes,
      public saga::monitorable
{
public:
    stream (saga::session const & s,
            saga::url          url = saga::url ());
    stream (saga::url          url);
    stream (void);
    stream (saga::object const & o);
    ~stream (void);

    stream & operator= (saga::object const & o);

    saga::url      get_url      (void) const;
    saga::context  get_context  (void) const;

    saga::context  connect      (void);

    std::vector <activity>
        wait      (activity      what,
                   double        timeout = -1.0);
    void          close      (double        timeout = 0.0);

    saga::ssize_t read      (saga::mutable_buffer buffer,
                             saga::ssize_t      length = 0);
    saga::ssize_t write     (saga::const_buffer  buffer,
                             saga::ssize_t      length = 0);
};

class server
    : public saga::object,
      public saga::monitorable,
      public saga::permissions
{
```

```
public:
server (saga::session const & s,
        saga::url          url = saga::url ());
server (saga::url          url);
server (void);
server (saga::object const & o);
~server (void);

server & operator= (saga::object const & o);

saga::url          get_url (void) const;
saga::stream::stream serve (double timeout = 0.0);
void              close (double timeout = 0.0);
};
}
```

## 12.3 Stream Details

The SAGA streams package allows establishing the simplest possible authenticated socket connection, with hooks to support application level authorization, and encryption schemes. The stream API has the following characteristics:

1. It is not performance-oriented: If performance is required, then it is better to program directly against the API's existing performance-oriented protocols like GridFTP or XIO.
2. It does not attempt to create a programming paradigm that diverges very far from baseline BSD sockets, Winsock, or Java Sockets.

This API greatly reduces the complexity of establishing authenticated socket connections in order to communicate with remotely located components. However, it provides very limited functionality and is thus suitable for applications that do not have very sophisticated requirements (as per 80-20 rule). It is envisaged that as applications become progressively more sophisticated, they will gradually move to more sophisticated native APIs in order to support those needs. Later SAGA versions may offer higher level communication abstractions, such as messages.

### 12.3.1 Endpoint URLs

The SAGA stream API uses URLs to specify connection endpoints. These URLs are supposed to allow SAGA implementations to be interoperable. For example,

the URL

```
tcp://remote.host.net:1234/
```

is supposed to signal that a standard `tcp` connection can be established with host `remote.host.net` on port 1234. No matter what the specified URL scheme is, the SAGA stream API implementation **MUST** have the same semantics on an API level, i.e., behave like a reliable byte-oriented data stream.

### 12.3.2 Usage

Just as for BSD sockets, a stream communication channel is established by creating a serving part (BSD: listening socket), and a client party (BSD: connecting socket). After connecting both parties, reading and writing on the stream allows to exchange data.

#### Stream example - server side

```
{
    // set up the serving endpoint on port 1234
    saga::stream::server server (saga::url ("tcp://localhost:1234"));

    // wait for incoming connections
    saga::stream stream = server.serve ();

    // if one arrived, greet the client
    stream.write (saga::buffer ("Hello World!", 13));
}
```

#### Stream example - client side

```
{
    // setup stream for the server endpoint
    saga::stream::stream (saga::url ("tcp://remotehost:1234"));

    // connect to the server
    stream.connect ();

    // read the greeting message
    saga::buffer buf (13);
    stream.read (buf);

    // print it
    std::cout << buf.get_data () << std::endl;
}
```

## 13 Using the RPC Package

### 13.1 Quick Introduction

Remote Procedure Calls are the second IPC mechanism provided in SAGA. The respective `saga::rpc` package is modeled after the GridRPC standard [?]. The concept could not be simpler: an `rpc` handle instance (`saga::rpc::rpc`) has a single method, `call()`, which invokes the respective remote operation. The call accepts a stack of `In`, `Out`, and `InOut` parameters.

### 13.2 Reference

Prototypes: `saga::rpc`

```
namespace saga
{
    namespace rpc
    {
        enum io_mode
        {
            Unknown = -1,
            In      = 1,
            Out     = 2,
            InOut   = In | Out
        };

        class parameter
            : public saga::mutable_buffer
        {
        public:
            parameter (void * data = 0,
                      saga::ssize_t size = -1,
                      io_mode mode = In,
                      buffer_deleter cb = default_buffer_deleter);

            ~parameter (void);

            io_mode get_mode() const;
        };

        class rpc
            : public saga::object,
              public saga::permissions
        {
        public:
            rpc (saga::session const & s,
```

```
saga::url          name = saga::url ();
rpc (std::string  const & name);
rpc (void);
~rpc (void);

void call (std::vector <parameter> parameters);
void close (double timeout = 0.0);
};
}
}
```

### 13.3 Details

The `rpc` class constructor is used to initialize the remote function handle. Be aware that this process may involve connection setup, service discovery, and other remote interactions! In the constructor, the remote procedure to be invoked is specified by a URL, with the syntax:

```
gridrpc://server.net:1234/my_function
```

with the elements responding to:

<code>gridrpc</code>	–	scheme	–	identifying a grid rpc operation
<code>server.net</code>	–	server	–	server host serving the rpc call
<code>1234</code>	–	port	–	contact point for the server
<code>my_function</code>	–	name	–	name of the remote method to invoke

All elements can be empty, which allows the implementation to fall back to invoke a default remote method.

Furthermore, the `rpc` class offers one method, `call()`, which invokes the remote procedure, and returns the respective `Out` and `InOut` parameters.

#### Remote Procedure Call Example

```
{
// initialize the rpc handle
saga::rpc::rpc rm(saga::url ("rpc://remote.host.net:31415/get_pi"));

// initialize the parameter stack
saga::rpc::parameter pi (NULL, -1, saga::rpc::Out);
std::vector <saga::rpc::parameter> parameters;
parameters.push_back (pi);
```



```
// invoke the remote procedure
rm.call (parameters);

// when completed, the output parameters are available
std::cout << "Pi equals " << pi.get_data () << std::endl;
}
```

The argument and return value handling is very basic, and reflects the traditional scheme for remote procedure calls, that is, an array of structures act as a parameter stack. For each element of the vector, the **parameter** struct describes its data **buffer**, the **size** of that buffer, and its input/output **mode**.

The **mode** value has to be initialized for each **parameter**, and the **size** and **buffer** values have to be initialized for each **In** and **InOut** struct. For **Out** parameters, **size** may have the value 0, in which case the **buffer** must be un-allocated. The **buffer** is to be created (e.g., allocated) by the SAGA implementation upon arrival of the result data, with a size that is sufficient to hold all result data. The **size** value is then set by the implementation to be equal to the allocated buffer size.

**HINT:**

This argument handling scheme allows efficient (copy-free) passing of parameters, because the parameter vector are passed by reference.

When an **Out** or **InOut** struct uses a pre-allocated buffer, any returned data exceeding the buffer size are discarded. The application is responsible for specifying correct buffer sizes for pre-allocated buffers; otherwise, the behaviour is in general undefined. For more details on buffer management, see Section ?? about buffer management.

## 14 Using the Advert Package

### 14.1 Quick Introduction

### 14.2 Reference

---

```

----- Prototypes: saga::advert -----
namespace saga
{
    namespace advert
    {
        namespace metrics
        {
            char const * const advert_modified
                = "advert.Modified";
            char const * const advert_deleted
                = "advert.Deleted";

            char const * const directory_created_entry
                = "directory.CreatedEntry";
            char const * const directory_modified_entry
                = "directory.ModifiedEntry";
            char const * const directory_deleted_entry
                = "directory.DeletedEntry";
        };

        enum flags
        {
            Unknown      = /* -1, */ saga::name_space::Unknown,
            None          = /* 0, */ saga::name_space::None,
            Overwrite     = /* 1, */ saga::name_space::Overwrite,
            Recursive     = /* 2, */ saga::name_space::Recursive,
            Dereference   = /* 4, */ saga::name_space::Dereference,
            Create        = /* 8, */ saga::name_space::Create,
            Exclusive     = /* 16, */ saga::name_space::Exclusive,
            Lock          = /* 32, */ saga::name_space::Lock,
            CreateParents = /* 64, */ saga::name_space::CreateParents,
                        // 256,      reserved for Truncate
                        // 512,      reserved for Append
            Read          = 512,
            Write         = 1024,
            ReadWrite     = 1536,
                        //2048,      reserved for Binary
        };

        class entry
        : public saga::name_space::entry,
          public saga::attributes
    }
}

```

---

```

{
    public:
        entry (saga::session const & s,
               saga::url          url,
               int                 mode = Read);
        entry (saga::url          url,
               int                 mode = Read);
        entry (saga::object const & other);
        entry (void);
        ~entry (void);

        entry & operator= (saga::object const & object);

        void          store_object    (saga::object object);
        saga::object retrieve_object (void);
        saga::object retrieve_object (saga::session const & s);

        void          store_string    (std::string str);
        std::string retrieve_string (void);
};

class directory
: public saga::name_space::directory,
  public saga::attributes
{
    public:
        directory (saga::session const & s,
                   saga::url          url,
                   int                 mode = ReadWrite);
        directory (saga::url url,
                   int                 mode = ReadWrite);
        directory (saga::object const & other);
        directory (void);
        ~directory (void);

        directory & operator= (saga::object const & o);

        advert::entry    open    (saga::url url,
                                   int        mode = Read);
        advert::directory open_dir (saga::url url,
                                    int        mode = Read);

        std::vector <saga::url>
            find    (std::string name,
                    std::vector <std::string> pattern,
                    int        flags = None)

};
}
}

```



### 14.3 Details

## Part III

# Advanced Topics

## 15 Using Asynchronous Operations

### 15.1 Quick Introduction

All functional SAGA API calls come in three flavours: synchronous, asynchronous, and task versions. The easiest way to understand the relations and differences between them is to consider that *all* functional API calls are represented by stateful **tasks** (e.g., a `file.copy()` API call is a task which can be in **Running** or **Done** state).

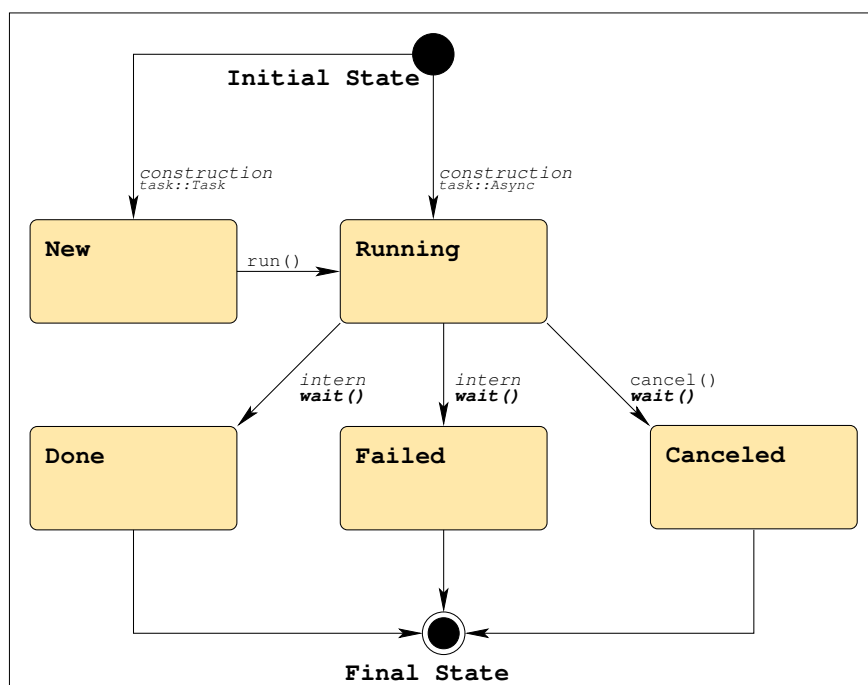


Figure 4: The SAGA task state model.

Figure ?? shows the available task states, and the various possible state transitions. The different SAGA API call invocations are simply representing ways to create the respective tasks in different states: Synchronous calls create tasks in a final state (**Done** or **Failed**); Asynchronous operations create tasks which are **Running**; and the Task version of the calls creates tasks which are **New**, and need to be `run()` to do anything at all:

#### Asynchronous operations

```
int main (int argc, char** argv)
{
```

```
// create a file instance
saga::url u (argv[1]);
saga::filesystem::file f (u);

// target dir to copy file to
saga::url tgt ("file://localhost/tmp");

// run file copy in three flavours
saga::task t_1 = f.copy <saga::task::Sync> (tgt);
saga::task t_2 = f.copy <saga::task::Async> (tgt);
saga::task t_3 = f.copy <saga::task::Task> (tgt);

// the three tasks do the same, but t_1 is already done at
// this point, t_2 could still be running, and t_3 did not yet
// start.

// let's get the async version done:
t_2.wait ();

// now, run and finish the Task version:
t_3.run ();
t_3.wait ();

// all three copies are done here.
}
```

For convenience, the synchronous version is provided simply as:

```
f.copy (tgt);
```

Now, what about the return values? For those, the tasks provide a typed member method, `get_result()`, which can only be called when a task is in a final state:

#### Return values for asynchronous operations

```
int main (int argc, char** argv)
{
    // create a file instance
    saga::url u (argv[1]);
    saga::filesystem::file f (u);

    // run method in three flavours
    saga::task t_1 = f.get_size <saga::task::Sync> ();
    saga::task t_2 = f.get_size <saga::task::Async> ();
    saga::task t_3 = f.get_size <saga::task::Task> ();

    // let's get the async version done:
```

```
t_2.wait ();

// now, run and finish the Task version:
t_3.run ();
t_3.wait ();

// all three calls are done here.

ssize_t size;
size = t_1.get_result <ssize_t> ();
size = t_2.get_result <ssize_t> ();
size = t_3.get_result <ssize_t> ();

}
```

The convenience of the synchronous version above is that it returns the return value immediately:

```
ssize_t size = f.get_size ();
```

## 15.2 Reference

## 15.3 Details



## 16 Using Monitorables and Notifications

### 16.1 Quick Introduction

#### Monitorables and Metrics

**FIXME:** fill in

#### Notifications

Closely related to monitorables and metrics are notifications; they notify the application of certain events. For example, when a task is done, when it changes its state, etc. For those events, the application can create and register custom callbacks:

Notifications on task state changes

```
#include <saga/saga.hpp>

class my_callback : public saga::callback
{
public:
    // cb exits the program with appropriate error code
    bool cb (saga::monitorable mt,
             saga::metric      m,
             saga::context      c)
    {
        if ( m.get_attribute ("state") == "Done" ) {
            exit (0);
        }

        if ( m.get_attribute ("state") == "Failed" ) {
            exit (-1);
        }

        // all other state changes are ignored
        return true; // keep callback registered
    }
}

int main (int argc, char** argv)
{
    // run a file copy asynchronously
    saga::url u (argv[1]);
    saga::filesystem::file f (u);
    saga::task t = f.copy <saga::task::Async> (saga::url (argv[2]));
}
```

```
// monitor the task state
my_callback cb;
t.add_callback ("state", cb);

// make sure the task finishes
t.wait ();
}
```

The example above registers a private callback to the **"state"** metric of the task. In fact, many SAGA objects (which implement the monitoring interface, and are thus **monitorables**) have several metrics, which can be queried by `monitorable.list_metrics()`.

**HINT:**

The return value of the callback determines if the callback stays registered (**true**) or not (**false**). This allows for a *call-once* semantics.

## 16.2 Reference

## 16.3 Details

## 17 Specifying Security Details

### 17.1 Quick Introduction

Tight security is arguably one of the most required features of Grid environments. On the other hand, for the average end user, it is also one of the most annoying features. That is mostly caused by the confusion about how security credentials are to be maintained, when and where they are valid, how to choose between them, etc.

SAGA can alleviate only some of these problems—more standardization work outside of SAGA is required to be able to simplify credential management further.

To understand SAGA's security model, one needs to consider two concepts: *sessions* and *contexts*; a `saga::context` simply represents one specific security credential; a `saga::session` can have multiple of these contexts attached, and is defined by the lifetime of the objects and operations using these credentials.

By default, both `saga::sessions` and `saga::contexts` are invisible: a default session is always implicitly created on the SAGA call, and picks up all the default security credentials your SAGA implementation knows about, such as your default ssh keys, your default globus proxy (if it was initialized before), your default Unicore keyring, etc. In most cases, that should allow writing applications which have no single line of code addressing security explicitly, which are still secure, as these credentials are used on adaptor level, as needed:

```
Secure gsiftp file copy
#include <saga/saga.hpp>

int main ()
{
    saga::url src ("ssh://remote.host.net/tmp/one.dat");
    saga::url tgt ("ssh://remote.host.net/tmp/two.dat");

    // do a file copy, using the default globus X509 proxy, in the
    // default session
    saga::filesystem::file f (src);
    f.copy (tgt);
}
```

On the other hand, it allows simple means of control on which credentials are to be used for a given operation, as in the following example:

## Secure gsiftp file copy

```
#include <saga/saga.hpp>

int main ()
{
    saga::url src ("ssh://remote.host.net/tmp/one.dat");
    saga::url tgt ("ssh://remote.host.net/tmp/two.dat");

    // create a new session, no default contexts are attached
    saga::session my_session;

    // create a new ssh context
    saga::context my_context ("ssh");

    // point ssh context to a specific ssh key
    my_context.set_attribute ("UserCert",
                             "/home/username/.ssh/id_dsa.special");

    // add that context to my session. The session will then
    // contain *only* that explicitly added context.
    my_session.add_context (my_context);

    // create a new file object, in that session
    saga::filesystem::file f (my_session, src);

    // do a file copy, using the specified ssh key
    f.copy (tgt);
}
```

**HINT:**

In most cases, the default session is what you want to use; if that does not work, try to convince your system administrator to configure SAGA so that the default session works!

Objects created from other objects inherit their session. Asynchronous operations are living in the session of their spawning objects. A session does not die when going out of scope, but only when all associated objects and operations die/finish.

## 17.2 Reference

## 17.3 Security Details

## 18 Miscellaneous Issues

### 18.1 Primitive Data Types

Alas, the C++ standard does not define all data types needed within SAGA. For that reason, we provide these types also in the `saga` namespace. Although these types should usually map to the native types on the respective platform the SAGA application is running on, we can only guarantee portability when using the SAGA provided types.

The types provided by SAGA are:

Primitive types in the `saga` namespace

```
namespace saga
{
    typedef ...   char_t;

    typedef ...   int8_t;
    typedef ...   uint8_t;

    typedef ...   int16_t;
    typedef ...   uint16_t;

    typedef ...   int32_t;
    typedef ...   uint32_t;

    typedef ...   int64_t;
    typedef ...   uint64_t;

    typedef ...   intmax_t;
    typedef ...   uintmax_t;

    typedef ...   long_long_t;

    typedef ...   size_t;
    typedef ...   ssize_t;
    typedef ...   off_t;

} // namespace saga
```

Note that the `int64_t` and `uint64_t` are only defined on those platforms which support 64 bit integers. How the individual types are defined is platform-dependent, and should not be relied upon. However, they will often be mapped to the respective Boost types.

## 18.2 Boost, and C++/TR1

Our implementation relies heavily on Boost [?]. Boost is an open source collection of state-of-the-art C++ libraries, is highly portable, and is actually thought to be the playground for upcoming versions of the C++ standard. In fact, all Boost features visible on the surface of our SAGA implementation (i.e., visible to the user of SAGA), are features which can already be found in the 'C++ Technical Recommendation 1' [?], a recent extension of the C++ standard. However, if your compiler does not support the TR1 extensions yet, Boost is required to use SAGA. Please refer to the installation manual for details on Boost.

**FIXME:** correct?