GFD-R-P.XX
SAGA-RG

Version: 1.0

Andre Merzky[1]

April 26, 2010

SAGA API Extension: Advert API

Status of This Document

This document provides information to the grid community, proposing a standard for an extension package to the Simple API for Grid Applications (SAGA). That extension provides access to persistent storage for serialized SAGA objects, and application level meta data (adverts). As SAGA extension, it depends upon the SAGA Core API Specification [2]. This document is supposed to be used as input to the definition of language specific bindings for this API extension, and as reference for implementors of these language bindings. Distribution of this document is unlimited.

Abstract

This document specifies an Advert API extension to the Simple API for Grid Applications (SAGA), a high level, application-oriented API for grid application development. This Advert API is motivated by a number of use cases collected by the OGF SAGA Research Group in GFD.70 [4], and by requirements derived from these use cases, as specified in GFD.71 [5]). It allows to persistently store application specific meta data in a name space hierarchy, along with serialized `saga::object` instances.

[1] editor

Contents

1 Introduction

A significant number of SAGA use cases [4] ask for the possibility to persistently store application level meta data[1]. In difference to data storage in files, these meta data are usually small, and structured as key-value-pairs. The main use case for this API extension is that an application stores some state information, and that these state information are either used by other applications, or by a later running instance of the same application.

For example, an application which allows to stream data (i.e. uses the SAGA Stream API [2]), may store its `saga::stream::service` endpoint URL as an advert, along with information about the protocol to be used, and another application which wants to connect to the first one may obtain the service object, and the protocol information, from the advert service. This allows, amongst others, for simple and environment independent bootstrapping of distributed ensembles of applications. The persistent nature of the advert service also allows applications to cooperate even if their actual application run time does not overlap.

Adverts are defined as an entry in the adverts name space, i.e. as an entry in an `saga::advert::directory`. Similar to `saga::replica::logical_file`, each advert can have meta data attached (i.e. has key-value based attributes). As described above, an `saga::advert` can also store one (serialized) `saga::object` instance. In some sense, that object instance can be considered to be the *content* of the advert, and the attributes can be considered the *meta data* of the advert, usually describing the content. Neither element needs to exist however – even completely empty adverts can be useful in some circumstances, e.g. to simply flag specific conditions.

1.1 Notational Conventions

In structure, notation and conventions, this documents follows those of the SAGA Core API specification [2], unless noted otherwise.

1.2 Security Considerations

As the SAGA API is to be implemented on different types of Grid (and non-Grid) middleware, it does not specify a single security model, but rather provides

[1] The distinction between data and meta data is usually not very well defined. In this document, we refer to meta data as small pieces of information which are used to manage the overall functionality of the application. They are, usually, not the data which are the object of the applications core algorithms. In particular, for the purpose of this document, we consider meta data *not* to be binary data.

hooks to interface to various security models – see the documentation of the `saga::context` class in the SAGA Core API specification [2] for details.

A SAGA implementation is considered secure if and only if it fully supports (i.e. implements) the security models of the middleware layers it builds upon, and neither provides any (intentional or unintentional) means to by-pass these security models, nor weakens these security models' policies in any way.

The implementations of advert services (the "backend" services to this API), need to take security concerns into account, because such a service might cause leaks of user (meta) data beyond the runtime of the applications using this API. This is the same risk as with storage and file systems, to which the SAGA core API provides an API. Unlike with established file systems, however, the risks associated with advert services might be less obvious to their implementors.

2 SAGA Advert API

2.1 Introduction

Several SAGA use cases [4], and also several current and past SAGA and GAT [1] base projects, declared the need for a simple interface to storage of small sets of persistent application data. Further, as distributed applications have an inherent need of coordination [3], the state for SAGA object instances is considered to count amongst those information. The advert API extension to SAGA, which is presented and specified in this document, is designed to accommodate those needs.

In its core, the advert package represents a `saga::namespace` derivate which allows to store, search and retrieve `saga::attribute` sets and `saga::object` derivates in its leave nodes. The notion of namespace is repeatedly used throughout the SAGA API [2], as is the notion of attributes. By combining both, the structure of the advert API package should actually be immediately clear. The novel addition to the package is the ability to store SAGA object instances, which should be considering as serialized representation of the the respective object's state.

The potential use cases of the API package are virtually endless, and as implementation of the API in SAGA and other APIs already exist since a number of years, the paradigm has already been proven to be incredibly useful for the development of distributed applications. An example applications is thus included to (a) demonstrate that usefulness, and (b) illustrate the structure and purpose of the API. The complete application code can be found in section 3.

Example: Master/Slave Application with Advert Registries

Assume a distributed application wants to employ the Master/Slave paradigm. The Master can then, after creating the slave jobs, publish those in a separate advert directory, which thus serves as this master's job registry. Each job advert contains the serialized job instance. Further, the master can publish work items in yet another advert directory, and assign job id's to each work item. That second advert directory acts as a work item queue. The work item adverts contain (a) a serialized SAGA file instance representing the work data, (b) the id of the job assigned to that work item, and (c) the state of that item (e.g. `'assigned'`). After all work items have been created and assigned, the jobs are `run()`, and can start to pick up work items.

The started slave processes search the work item registry for items assigned to them, by doing a `find()` on the advert directory, with a pattern which specifies `'work_id=<my_id>'`, with `my_id` being their own job id. They then

work on each item, marking it as `'accepted'` when starting the work, and as `'completed'` when done.

A separate master process could decide to check the overall progress of the work. To do that, it retrieves all job and work item adverts, and checks the respective status: for the jobs, it retrieves the job instances from the job adverts, and calls `get_state()` on them; for the work items, it checks the `'work_state'` attribute of the work item adverts. If jobs are in a final state, and all work items are completed, the master can safely purge the advert directories.

That example obviously is very simplistic in respect to scheduling of work items, and also in respect to error recovery, but is nevertheless fully functional. Creating an application with similar functionality without the help of the advert service requires significantly more, and also more complex, operations. In particular, the application is immediately resilient against master failures: once the job and work item registries exist, they are persistent, and can be utilized by any application component with the respective permissions. Further, the communication between the individual application components (i.e. processes) is immediately asynchronous, secure, and persistent (no 'messages' get lost). Also, the registries allow to easily infer the overall state of the distributed application. Finally, the communication via the advert service completely solves the application bootstrapping problem: there is no need for any application component to directly contact any other component. Thus, no component needs to know where any other component is actually being executed. The only shared information are the URLs of the job and work item registries (or, in our code base, the single URL of the directory containing these registries).

The complete C++ source code for the example described above is listed in appendix 3, along with additional shorter code examples.

2.1.1 Classes

The SAGA Advert API consists of two classes: the `advert::advert` class, which inherits `namespace::entry` and encapsulates the application information to be stored persistently; and the `advert::directory` class, which inherits the `namespace::directory` and represents the directory adverts are organized in. The `advert::advert` class has three additional methods, `store_object()`, `retrieve_object()` and `delete_object()`, which allow to associate a SAGA object instance with that specific advert. SAGA object instances of those classes which are defined in either the SAGA Core API Specification [2], or in one of the specifications for SAGA API extension packages. In either case, the classes to be serialized MUST inherit the `saga::object` interface from [2].

Storing and retrieving a SAGA object is semantically equivalent to the `object.clone()` call, as specified in [2], with the only difference that the cloning can now poten-

tially span over completely independent and different application instances.

The `advert::directory` has an overloaded `find()` method, which allows to also search object types, and for meta data pattern (i.e. attribute patterns), similar to the find of the SAGA replica package. Additionally, the `advert::flags` enum is inherited from the SAGA namespace package, and extended by the `Truncate` flag which empties both the associated object and the attributes of the advert to be opened.

Note that the `advert.retrieve_object()` method is able to return different object types. It thus uses the same type templatization signature as employed in the SAGA core specification, for example for the `task.get_result()` method. Language bindings MAY utilize the same technique for `advert.store_object()`, if the argument's type cannot automatically inferred in that language.

2.1.2 Advert State Attributes and Object Serialization

As SAGA is an API specification, it is generally true that interoperability on backend level can neither be specified, nor enforced, by SAGA. This document is thus silent about the exact mechanism used to implement the object serialization, and its representation in the backend. It is clear, however, that the implementation MUST ensure that enough information are stored so that an equivalent object can be recreated when calling `retrieve_object()`.

SAGA objects usually live in a specific SAGA session, which has a set of associated SAGA contexts. The serialization MUST NOT attempt to serialize session and contexts. Instead, the objects get deserialized in the session of the deserializing advert instance. That may result in objects which cannot function due to missing security contexts. A session serialization could, however, also not guarantee functional credentials. Further, both the package semantics and the usability would be significantly complicated when attempting to cover session and context serialization automatically.

For those use cases where session and context persistence is essential, that semantics can always achieved manually, by

- serializing the required contexts,
- creating an empty session on deserialization side,
- filling that session with deserialized contexts,
- opening adverts in that session,
- retrieving then the required SAGA objects from that advert.

While that process seems tedious, it seamlessly fits the overall advert package semantics. We also believe that only a minority of use cases require that semantics.

2.1.3 Advert Persistency and Lifetime Management

Adverts have, by default, an unspecified lifetime, and can thus in particular survive the application which created the advert. It should be noted that this can, however, lead to garbage, i.e. to an increasing number of entries which are not needed anymore. To support user and system level garbage collection, the `set_ttl (int)` method on the `advert::entry` and `advert::directory` classes can be used to specify a minimal advert lifetime (time-to-live, ttl) – beyond that time, the advert or advert directory will be considered as expired, or garbage, and MUST be discarded by the backend.

Note that, as advert directories and entries inherit from the `saga::namespace` package from [2], they both have a `last_modified` property. In general it should hold that the expiration time equals the `last_modified` time plus the specified ttl.

If the ttl of an open advert or advert directory is expired, the result of any call accessing that advert MUST throw a `'IncorrectState'` exception. Any attempt to open an expired advert MUST result in a `'DoesNotExist'` exception.

If no ttl is defined on an advert or advert directory, it is assumed to never to expire.

2.1.4 Advert URLs

The exact rendering of the advert namespace is up to the respective implementation, and it is thus not specified in this document how valid URLs are formed (i.e. what schemas are supported). Implementations SHOULD, however, strive to support the generic URL schema 'any', as motivated in [2]. Otherwise, the rules specified for file system URLs in [2] SHOULD be followed.

2.1.5 Implementation Interoperability

The SAGA is, as API specification, generally silent about backend interoperability. We expect however, that implementations of the advert API extension can potentially be interoperable, even across different programming languages, in the sense that adverts attributes and associated SAGA objects can be stored in one implementation, in one programming language, and be retrieved completely, and as fully functional SAGA object instances, in another programming language. One way to achieve that interoperability would be to specify a serialization scheme, and to define the minimal set of object state attributes required to reinstantiate a SAGA object. While that is certainly possible, it is out of scope for this document, and should be addressed in a different specification.

2.2 Specification

```
package saga.advert
{
  enum flags : extends saga::namespace::flags
  {
    None            =    0,   // from saga::namespace
    Overwrite       =    1,   // from saga::namespace
    Recursive       =    2,   // from saga::namespace
    Dereference     =    4,   // from saga::namespace
    Create          =    8,   // from saga::namespace
    Exclusive       =   16,   // from saga::namespace
    Lock            =   32,   // from saga::namespace
    CreateParents   =   64,   // from saga::namespace
    Truncate        =  128,
    Read            =  512,   // from saga::namespace
    Write           = 1024,   // from saga::namespace
    ReadWrite       = 1536    // from saga::namespace
  }


  class directory : extends saga::ns_directory
                    extends saga::attributes
      // from ns_directory saga::ns_entry
      // from ns_entry     saga::object
      // from ns_entry     saga::async
      // from ns_entry     saga::permissions
      // from object       saga::error_handler
  {
    CONSTRUCTOR   (in  session            session,
                   in  string             url,
                   in  int                flags = Read,
                   out directory          obj);
    DESTRUCTOR    (in  directory          obj);

    // set/get time to live
    set_ttl       (in  uint               ttl);
    get_ttl       (out uint               ttl);

    // find adverts based on name, object type, and meta data
    find          (in  string             name_pattern,
                   in  array<string>      attr_pattern,
                   in  saga::object::type type  = 0,
                   in  int                flags = Recursive,
```

```
                    out array<saga::url>   names );

      // Attributes (extensible):
    }


    class entry : extends        saga::ns_entry
                  extends        saga::attributes
              // from ns_entry  saga::object
              // from ns_entry  saga::async
              // from ns_entry  saga::permissions
              // from object    saga::error_handler
    {
      CONSTRUCTOR     (in    session       session,
                       in    string        url,
                       in    int           flags = Read,
                       out   entry         obj);
      DESTRUCTOR      (in    entry         obj);

      // set/get time to live
      set_ttl         (in    uint          ttl);
      get_ttl         (out   uint          ttl);

      // attach saga::object instances
      store_object    (in    saga::object  content);
      retrieve_object <type>
                      (out   saga::object  content);
      delete_object   (void);

      // Attributes (extensible):
    }
  }
```

2.3 Specification Details

2.3.1 Enum `flags`

The flags describe the properties of several operations on advert directories and entries. This package inherits the flags from the namespace package, and uses the same flag semantics unless specified otherwise. The `Truncate` flags is added, which is to be used when opening an `advert::entry` instance shall completely empty that entry. The `Truncate` flag does not imply a reset of the creation time, but it causes the entry's ttl counter to be restarted. On advert directories, the

`Truncate` flags causes the attributes on that directory instances to be purged, and any associated SAGA object instance to be removed, but leaves the entries and subdirectories of this instance untouched.

2.3.2 Class `advert::directory`

The `advert::directory` class follows the purpose and semantics of the inherited `saga::namespace::directory` class. It additionally inherits the `saga::attribute` interface, which allows the SAGA user to store arbitrary key-value pairs on the advert directory.

The class has two additional methods to query and set the directory's ttl. If that time is passed (i.e. the directory's creation-time plus its ttl is smaller than 'now'), the directory is considered to be expired. It MUST, however, be kept in a usable state as long as it (or its subdirs) contains any entries which are not expired. The ttl counter (re)starts on creation time, whenever the directory is being modified (i.e. when any directory attributes are changed, when entries or subdirectories are added, renamed or removed), and when calling `set_ttl()`.

Another namespace method, `find()`, is overloaded, and allows to extends the search pattern to (a) the type of objects associated with adverts, and (b) the attributes associated with adverts.

```
- CONSTRUCTOR
  Purpose:  create the object
  Format:   CONSTRUCTOR          (in  session   s,
                                  in  saga::url name,
                                  in  int       flags = Read,
                                  out directory obj)
  Inputs:   s:                    session handle
            name:                 location of directory
            flags:                open mode
  InOuts:   -
  Outputs:  obj:                  the newly created object
  PreCond:  -
  PostCond: - the directory is opened.
            - 'Owner' of directory is the id of the context
              used to perform the operation, if the
              directory gets created.
            - the ttl timer of the object is started on
              Creation, and if the Truncate flag is
              specified.
  Perms:    Exec  for parent directory.
```

```
            Write for parent directory if Create is set.
            Write for name if Write is set.
            Read  for name if Read  is set.
  Throws:   NotImplemented
            IncorrectURL
            BadParameter
            DoesNotExist
            AlreadyExists
            PermissionDenied
            AuthorizationFailed
            AuthenticationFailed
            Timeout
            NoSuccess
  Notes:    - if the 'Truncate' flag is given, the returned
              object MUST NOT have an associated object, and
              MUST have an empty attribute set.
            - the 'Truncate' flag requires that the entry
              exists, or that the 'Create' flag is given,
              too.  Otherwise, a DoesNotExist exception is
              thrown.
            - the 'Create' flag implies 'Write'.


- DESTRUCTOR
  Purpose:  destroy the object
  Format:   DESTRUCTOR           (in  entry        obj)
  Inputs:   obj:                  the object to destroy
  InOuts:   -
  Outputs:  -
  PreCond:  -
  PostCond: - the directory is closed.
  Perms:    -
  Throws:   -
  Notes:    -


- set_ttl
  Purpose:  set a time to life, and restart the ttl timer.
  Format:   set_ttl              (in   uint  ttl);
  Inputs:   ttl:                  time to live in seconds
  InOuts:   -
  Outputs:  -
  PreCond:  -
  PostCond: - the instance's ttl timer is restarted.
            - the instance's ttl is set to ttl.
  Perms:    - Write
```

```
  Throws:   NotImplemented
            IncorrectState
            BadParameter
            Timeout
            NoSuccess
  Notes:    - A ttl value '0' declares the instance as
              garbage immediately.
            - backends MAY decline specific TTL parameter,
              if they are not willing to guarantee that
              lifetime.  In those cases, the implementation
              MUST throw a 'BadParameter' exception.


- get_ttl
  Purpose:  get the time to life
  Format:   get_ttl               (out  uint  ttl);
  Inputs:   -
  InOuts:   -
  Outputs:  ttl:                  time to live in seconds
  PreCond:  -
  PostCond: - the instance's ttl timer not restarted.
  Perms:    - Read
  Throws:   NotImplemented
            IncorrectState
            Timeout
            NoSuccess
  Notes:    -


- find
  Purpose:  find adverts in the current directory and below,
            with matching names and matching meta data
  Format:   find  (in  string             name_pattern,
                   in  array<string>      attr_pattern,
                   in  saga::object::type type  = 0,
                   in  int                flags = Recursive,
                   out array <saga::url>  names);
  Inputs:   name_pattern:     pattern for names of
                              entries to be found
            attr_pattern:     pattern for meta data
                              key/values of entries to be
                              found
            type:             filter for adverts with
                              attached saga objects of that
                              type
            flags:            flags defining the operation
```

```
                              modus
InOuts:   -
Outputs:  names:              array of names matching all
                              criteria
PreCond:  -
PostCond: -
Perms:    Read  for cwd.
          Query for entries specified by name_pattern.
          Exec  for parent directories of these entries.
          Query for parent directories of these entries.
          Read  for directories specified by name_pattern.
          Exec  for directories specified by name_pattern.
          Exec  for parent directories of these directories.
          Query for parent directories of these directories.
Throws:   NotImplemented
          BadParameter
          IncorrectState
          PermissionDenied
          AuthorizationFailed
          AuthenticationFailed
          Timeout
          NoSuccess
Notes:    - the semantics for both the find_attributes()
            method in the saga::attributes interface and
            for the find() method in the
            saga::ns_directory class apply.  On
            conflicts, the find() semantic supersedes
            the find_attributes() semantic.  Only entries
            matching all attribute patterns, the name
            space pattern and the object type are returned.
          - the default flags are 'Recursive' (2).
          - expired entries (see ttl) MUST NOT be returned.
```

2.3.3 Class `advert::advert`

The `advert::advert` class follows the purpose and semantics of the inherited `saga::namespace::entry` class. Two methods allow to manage the `saga::object` instance associated with that advert entry.

Advert entry instances also have a ttl, which follows the same semantics as defined above for the advert directory. Further, the advert entry implements the `saga::attributes` interface, and can thus hold an arbitrary set of user defined attributes.

```
- CONSTRUCTOR
  Purpose:  create the object
  Format:   CONSTRUCTOR          (in  session   s,
                                  in  saga::url name,
                                  in  int       flags = Read,
                                  out entry    obj)
  Inputs:   s:                    session handle
            name:                 initial working dir
            flags:                open mode
  InOuts:   -
  Outputs:  obj:                  the newly created object
  PreCond:  -
  PostCond: - the entry is opened.
            - 'Owner' of target is the id of the context
              use to perform the operation, if the
              entry gets created.
  Perms:    Exec  for parent directory.
            Write for parent directory if Create is set.
            Write for name if Write is set.
            Read  for name if Read  is set.
  Throws:   NotImplemented
            IncorrectURL
            BadParameter
            DoesNotExist
            AlreadyExists
            PermissionDenied
            AuthorizationFailed
            AuthenticationFailed
            Timeout
            NoSuccess
  Notes:    - semantic as in saga::namespace::entry
            - if the 'Truncate' flag is given, the returned
              object MUST NOT have an associated object, and
              MUST have an empty attribute set.
            - the 'Truncate' flag requires that the entry
              exists, or that the 'Create' flag is given,
              too.  Otherwise, a DoesNotExist exception is
              thrown.
            - the 'Create' flag implies 'Write'.


- DESTRUCTOR
  Purpose:  destroy the object
  Format:   DESTRUCTOR           (in  entry        obj)
```

```
  Inputs:   obj:                  the object to destroy
  InOuts:   -
  Outputs:  -
  PreCond:  -
  PostCond: - the entry is closed.
            - the instance's ttl timer is not restarted.
  Perms:    -
  Throws:   -
  Notes:    - semantic as in saga::namespace::entry


- set_ttl
  Purpose:  set a time to life, and restart the ttl timer.
  Format:   set_ttl               (in  uint  ttl);
  Inputs:   ttl:                  time to live in seconds
  InOuts:   -
  Outputs:  -
  PreCond:  -
  PostCond: - the instance's ttl timer is restarted.
            - the instance's ttl is set to ttl.
  Perms:    - Write
  Throws:   NotImplemented
            IncorrectState
            Timeout
            NoSuccess
  Notes:    - all notes to advert::directory::set_ttl()
              method apply


- get_ttl
  Purpose:  get the time to life
  Format:   get_ttl               (out  uint  ttl);
  Inputs:   ttl:                  time to live in seconds
  InOuts:   -
  Outputs:  -
  PreCond:  -
  PostCond: - the instance's ttl timer is not restarted.
  Perms:    - Read
  Throws:   NotImplemented
            IncorrectState
            Timeout
            NoSuccess
  Notes:    - all notes to advert::directory::get_ttl()
              method apply
```

```
- store_object
  Purpose:  associate a saga::object instance with the entry
  Format:   store_object          (in  saga::object content);
  Inputs:   content:              saga::object to be
                                  associated with the entry
  InOuts:   -
  Outputs:  -
  PreCond:  -
  PostCond: - the given object instance can be retrieved
              with retrieve_object().
            - any reference to an previously associated
              object is removed.
            - the advert's ttl is reset
  Perms:    -
  Throws:   NotImplemented
            IncorrectState
            Timeout
            BadParameter
            NoSuccess
  Notes:    - if the implementation does not support the
              association of that object type, a
              'BadParameter' exception is thrown.
            - if no object is associated with this advert,
              an 'IncorrectState' exception is thrown.


- retrieve_object
  Purpose:  retrieve the associated saga::object instance
  Format:   retrieve_object       (out saga::object content);
  Inputs:   -
  InOuts:   -
  Outputs:  content:              saga::object associated
                                  with the entry
  PreCond:  -
  PostCond: -
  Perms:    -
  Throws:   NotImplemented
            IncorrectState
            Timeout
            BadParameter
            NoSuccess
  Notes:    - if the implementation cannot de-serialize the
              stored object type, a 'NoSuccess' exception is
              thrown.  Language bindings MAY throw a native
              type mismatch exception.
            - if no object is associated with this advert,
```

```
              an 'IncorrectState' exception is thrown.
            - if the implementation can deserialize the
              stored object type, but cannot deserialize
              that specific instance, an 'IncorrectState'
              exception is thrown.
            - the object stays associated with the entry.
            - each call to this method retrieves a new copy
              of the original object.
            - for all practical purposes, retrieve_object
              behaves exactly like object.clone().


- delete_object
  Purpose:  de-associate a saga::object instance from the entry
  Format:   delete_object          (void);
  Inputs:   -
  InOuts:   -
  Outputs:  -
  PreCond:  -
  PostCond: - the given object instance cannot be retrieved
              with retrieve_object() anymore.
            - the advert's ttl is reset
  Perms:    -
  Throws:   NotImplemented
            IncorrectState
            Timeout
            NoSuccess
  Notes:    - if no object is associated with this advert,
              an 'IncorrectState' exception is thrown.
```

3 Example Code

This section lists a number of C++ code examples, to illustrate the use of the Advert API package. These examples are **not normative**.

3.1 Advertising Jobs

This first example code runs a simple job, and creates an advert entry to publish the created job instance. The second code snippet, representing a different SAGA application, scans the used advert directory for all entries, and prints some details for all jobs found there. A real world application would need to ensure that the used advert entry names are unique, for example basing them on the job id.

Advertising Jobs

```
1  // run a simple job
2  saga::job::service js;
3  saga::job::job j = js.run_job ("sleep 1000");
4
5  // create an advert for that job, and publish the job instance
6  saga::advert::entry advert ("any:///users/merzky/jobs/test.adv",
7                              saga::advert::Create);
8  advert.store_object (j);
```

Scanning Published Jobs

```
1  // open the resource job directory
2  saga::advert::directory adir ("any:///users/merzky/jobs/");
3
4  // list all registered (i.e. advertised) jobs
5  std::vector <saga::url> ads = adir.list ();
6
7  // for each found job, show some basic info
8  for ( unsigned int i = 0; i < ads.size (); i++ )
9  {
10   saga::advert::entry ad = adir.open (ads[i], saga::advert::Read);
11   saga::job::job j = ad.retrieve_object <saga::job::job> ();
12
13   std::cout << "id     : " << j.get_job_id () << std::endl
14             << "state  : " << j.get_state  () << std::endl
15             << "exe    : " << j.get_description ()
16                               .get_attribute ("Executable")
17             << std::endl << std::endl;
18 }
```

3.2 Advertising Resources

The first code snippet advertises information about the compute resource it is running on. Those information could, for example, be used for application level scheduling decisions. Pseudo functions are used to obtain and parse the output of Unix command line commands. Also, we use pseudo string concatenation (`"abc" + "def"`) to further simplify the example.

Advertising Resources

```
1  // get the name of the host we run on
2  std::string hostname = run ("hostname");
3
4  // create an advert for this resource, if it doesn't yet exist
5  saga::advert::entry ad (any:///users/merzky/resources/" + hostname,
6                          saga::advert::Create);
7
8  // publish some resource attributes
9  ad.add_attribute ("hostname"      , cut  (-1,  // last field
10                                       grep ("^" + hostname,
11                                       run  ("dig " + hostname))));
12 ad.add_attribute ("ostype"        , run  ("uname -s"));
13 ad.add_attribute ("architecture"  , run  ("uname -m"));
14 ad.add_attribute ("n_cpus"        , wc   ("-l",
15                                       grep ("^processor,
16                                       run  ("cat /proc/cpuinfo"))));
17 ad.add_attribute ("size_home"     , cut  (2,
18                                       run  ("df -h ~")));
19 ad.add_attribute ("size_tmp"      , cut  (2,
20                                       run  ("df -h /tmp/")));
21 ad.add_attribute ("size_data"     , cut  (2,
22                                       grep ("^Mem:",
23                                       run  ("free -g"))));
24 ad.add_attribute ("load"          , grep ("^Cpu",
25                                       run  ("top -b -n1")));
```

The resource adverts published by the example above can be used to run an application on a specific CPU architecture, for example.

Using Resource Adverts

```
1  // open the resource advert directory
2  saga::advert::directory adir ("any:///users/merzky/resources/");
3
4  // find resources adverts with matching attributes
5  std::vector <saga::url> candidates =
6      adir.find ("*",                // all resources
7                 "architecture=x68*" // with an x86 arch
8                                     // (x86, x86_32, x86_64, ...)
9                );
10
11 // ensure we found a matching resource
12 assert ( candidates.size () > 0 );
13
14 // run the job on the first matching resource
15 saga::job::service js;
16 js.run_job ("/path/to/application", candidates[0]);
```

3.3 Master-Worker Example

For a high level description of this example application, see section 2.1.

Master Code - Startup

```
1
2   #define BASE_URL std::string ("any://advert.db.net/my_app")
3   #define JOBNUM   100  // size of worker pool
4   #define WORKNUM  1000 // number of work items
5
6   // the master spawns jobs, and assigns them work items.  These info
7   // are stored in the advert service, waiting for the jobs to pick
8   // them up, and report back.
9   int main ()
10  {
11    // a job description - details are left to the reader
12    saga::job::description jd;
13
14    // create the job service used to spawn the slaves
15    saga::job::service js ("any://job.service.net");
16
17    // create the job registry in the advert data base
18    saga::advert::directory jobs (BASE_URL + "jobs/",
19                                  saga::advert::Create);
20
21    // keep track of jobs and job_ids
22    saga::task_container tc;
23    std::vector <std::string> job_ids;
24
25    // spawn the slaves
26    for ( int i = 0; i < JOBNUM; i++ )
27    {
28      saga::job::job j = js.create_job (jd);
29
30      // register the slaves in the registry
31      saga::entry a = jobs.open (j.get_jobid (),
32                                 saga::advert::Create);
33      a.store_object (j);
34
35      // keep job and jobid
36      tc.add_task (j);
37      job_ids.push_back (j.get_jobid ());
38    }
39
40    // create the work item registry in the advert data base
41    saga::advert::directory works ("BASE_URL + "works/",
42                                   saga::advert::Create);
43
44    // publish work items, and assign them to the slaves
```

```
45    for ( int i = 0; i < WORKNUM; i++ )
46    {
47      // open file representing the work item (pseudo code)
48      saga::filesystem::file f ("any://data.src.net/data/set_[i].dat");
49
50      // publish it in the work item queue
51      saga::entry a = works.open (f.get_name (),
52                                  saga::advert::Create);
53      a.store_object (f);
54
55      // assign it to a job (pseudo code)
56      a.set_attribute ("worker_id",    job_ids[j % JOBNUM]);
57      a.set_attribute ("worker_state", "assigned");
58    }
59
60    // work items are created and assigned, now we can start the jobs,
61    // so that they can begin to pick up work
62    tc.run ();
63
64    // the master can safely exit here, as all job and work item info
65    // are persistently stored in the advert service
66    return 0;
67  }
```

Client Code Code - Work

```
1  #define BASE_URL std::string ("any://advert.db.net/my_app")
2
3  // the client gets its own job_id, and retrieves all work items
4  // assigned to it.  After completing them, it ticks them off in the
5  // registry, and finishes if no further work is pending.
6  int main ()
7  {
8    // get own job id
9    saga::job::service js;
10   saga::job::job     me = js.get_self  ();
11   std::string        id = me.get_jobid ();
12
13   // retrieve a data items from the work item queue
14   saga::advert::directory works (BASE_URL + "works/");
15
16   std::vector <std::string> pat;            // meta data to match this
17   pat.push_back ("worker_id=" + id);        // pseudo code string ops
18   pat.push_back ("worker_state=assigned");  // only pick new items
19
20   // this worker type can only work on files
21   std::vector <saga::url> items = works.find ("*", pat,
22                                               saga::object::File);
23   while ( ! items.empty () )
24   {
25     // work on the items
26     for ( int i = 0; i < items.size (); i++ )
27     {
28       // open the work item
29       saga::advert::entry a = works.open (items[i]);
30
31       // signal that we work on that item
32       a.set_attribute ("worker_state", "accepted");
33
34       // do work, on the file which is 'contained' in the advert
35       do_work (a.get_object <saga::filesystem::file> ());
36
37       // signal that item is completed
38       a.set_attribute ("worker_state", "completed");
39     }
40
41     // refresh work item list
42     items = works.find ("*", pat, saga::object::File);
43   }
44
45   // done - just finish
46   return 0;
47 }
```

Master Code - Check and Finish

```
1  #define BASE_URL std::string ("any://advert.db.net/my_app")
2
3  // another master (yes, we have two) checks the status of jobs and
4  // workers, and cleans up if everything is done.
5  int main ()
6  {
7    bool completed = true;
8
9    // open the work item registry in the advert data base, and get
10   // all work items
11   saga::advert::directory works (BASE_URL + "works/");
12   std::vector <saga::url>  items = works.list ();
13
14   // check item state
15   for ( int i = 0; i < items.size (); i++ )
16   {
17     saga::advert::entry a = works.open (items[i]);
18     std::cout << " item "       << i
19               << " handled by " << a.get_attribute ("worker_id")
20               << " has state "  << a.get_attribute ("work_state")
21               << std::endl;
22
23     // check global state
24     if ( a.get_attribute ("work_state") != "completed" )
25     {
26       completed = false;
27     }
28   }
29
30
31   // open the job registry in the advert data base, and get all jobs
32   saga::advert::directory jobs (BASE_URL + "jobs/");
33   std::vector <saga::url>  ids = jobs.list ();
34
35   // check item state
36   for ( int i = 0; i < ids.size (); i++ )
37   {
38     saga::advert::entry a = jobs.open (ids[i]);
39     saga::job::job      j = a.get_object <saga::job::job> ();
40
41     std::cout << " job "       << i
42               << " has id "    << ids[i]
43               << " and state " << j.get_attribute ("State")
44               << std::endl;
45
46     // check global state
47     if ( j.get_state != saga::job::Done   ||
48          j.get_state != saga::job::Failed )
```

```
49      {
50        completed = false;
51      }
52    }
53
54
55    // if everything is done, we can clean up the advert service dirs.
56    // Otherwise, we just wait for the next run to do so, eventually.
57    if ( completed )
58    {
59      works.remove (saga::advert::Recursive);
60      jobs.remove  (saga::advert::Recursive);
61    }
62
63    return (completed ? 0 : 1);
64  }
```

4 Intellectual Property Issues

4.1 Contributors

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

Andre Merzky
andre@merzky.net
Center for Computation and Technology
Louisiana State University
216 Johnston Hall
70803 Baton Rouge
Louisiana, USA

The initial version of the presented SAGA API was drafted by members of the SAGA Research Group. Members of this group did not necessarily contribute text to the document, but did contribute to its current state. Additional to the authors listed above, we acknowledge the contribution of the following people, in alphabetical order:

Andrei Hutanu (LSU), Hartmut Kaiser (LSU), Pascal Kleijer (NEC), Thilo Kielmann (VU), Gregor von Laszewski (ANL), Shantenu Jha (LSU), John Shalf (LBNL), and Ole Weidner (LSU).

4.2 Intellectual Property Statement

The OGF takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the OGF Secretariat.

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

4.3 Disclaimer

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

4.4 Full Copyright Notice

References

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

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

[3] S. Jha, D. S. Katz, M. Parashar, O. Rana, and M. Cole. Abstractions for distributed systems (dpa 2008). In Euro-Par Workshops, page 401, 2008.

[4] A. Merzky and S. Jha. A Collection of Use Cases for a Simple API for Grid Applications. Grid Forum Document GFD.70, 2006. Global Grid Forum.

[5] A. Merzky and S. Jha. A Requirements Analysis for a Simple API for Grid Applications. Grid Forum Document GFD.71, 2006. Global Grid Forum.