Project no. IST-033576

XtreemOS

Integrated Project
BUILDING AND PROMOTING A LINUX-BASED OPERATING SYSTEM TO SUPPORT VIRTUAL ORGANIZATIONS FOR NEXT GENERATION GRIDS

First Prototype of XtreemOS Runtime Engine
D3.1.3

Due date of deliverable: November 30^{th}, 2007
Actual submission date: December 17^{th}, 2007

Start date of project: June 1^{st} 2006

Type: Deliverable
WP number: WP3.1
Task number: T3.1.2

Responsible institution: VUA
Editor & and editor's address: Thilo Kielmann
Vrije Universiteit
Dept. of Computer Science
De Boelelaan 1083
1081HV Amsterdam
The Netherlands

Version 1.0 / Last edited by Thilo Kielmann / December 17, 2007

Project co-funded by the European Commission within the Sixth Framework Programme		
Dissemination Level		
PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Revision history:

Version	Date	Authors	Institution	Section affected, comments
0.9	30/10/07	Andre Merzky and Thilo Kielmann	VUA	initial draft
0.99	05/11/07	Mathijs den Burger	VUA	ready for review
1.0	17/12/07	Thilo Kielmann	VUA	final version, based on reviewer comments

Reviewers:
Josep M. Perez Cancer (UPC), Matthias Hess (NEC)

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
T3.1.2	A runtime engine for dynamic call dispatching	VUA*

°This task list may not be equivalent to the list of partners contributing as authors to the deliverable.
*Task leader

Executive Summary

This document presents the first prototype implementation of a runtime engine for XtreemOS API as specified in deliverable D3.1.1 [6]. We outline the design of the implementation, explain how to install and deploy it, and refer to a programmer's manual.

1 Introduction

For the successful adoption of the XtreemOS grid operating system, it is extremely important to provide a well-accepted API to its potential application programs. To accomplish this goal, we are following an iterative approach to specifying and implementing this API. In our previous deliverable, D3.1.1. [6], we have presented the *Simple API for Grid Applications (SAGA)* [2] as the first draft API for XtreemOS.

In this deliverable, we provide a first prototype implementation of the SAGA API, written in the C++ programming language. In a companion deliverable, D3.1.2 [7], we provide the second draft API for XtreemOS, focusing on SAGA extensions to accomodate specific XtreemOS features. Those extensions will be provided in future implementation releases.

In this document, we outline the underlying design principles of our SAGA C++ implementation, and provide information for its download, installation, and use.

2 The XtreemOS API, first draft specification

In our previous deliverable D3.1.1 [6], we have motivated the use of the SAGA API [2] specification as *the first draft XtreemOS API*. That deliverable also contains a brief overview of the SAGA API, its structure and its scope. Since the publication of D3.1.1, the SAGA API specification evolved significantly, but mostly in terms of consistency and look & feel rather than in terms of scope and functionality. Notably, several changes proposed by XtreemOS partners have been incorporated, amongst them a different mechanism for asynchronous method invocation, and the inclusion of the JSDL SPMD extension [5], both essential to the XtreemOS environment.

In general, the observations made in D3.1.1 still hold: the SAGA API covers the scope of XtreemOS very well, with only very few notable exceptions such as checkpoint and recovery, and ACLs according to the withdrawn POSIX draft IEEE 1003.1e/2c. Also, our implementation of the SAGA API in C++ (described in this document) shows that (a) the API can sensibly be implemented, and (b) provides the promised simplified access to grid resources.

3 A runtime engine for the first draft XtreemOS API

In its general architecture, our SAGA implementation follows the lessons we have learned with the SAGA predecessor GAT [1]: a small dynamic *engine* provides

dynamic call switching of SAGA API calls to middleware bindings (*adaptors*) which are dynamically loaded on demand, and bound at runtime (*late binding*). The relation between these components are illustrated in Figure 1. Unlike the GAT, SAGA provides an extensible API framework, consisting of a look & feel part, and an extensible set of functional packages. Implementations need to take special care of this extensibility. We explain our approach to an extensible implementation in the engine description below.

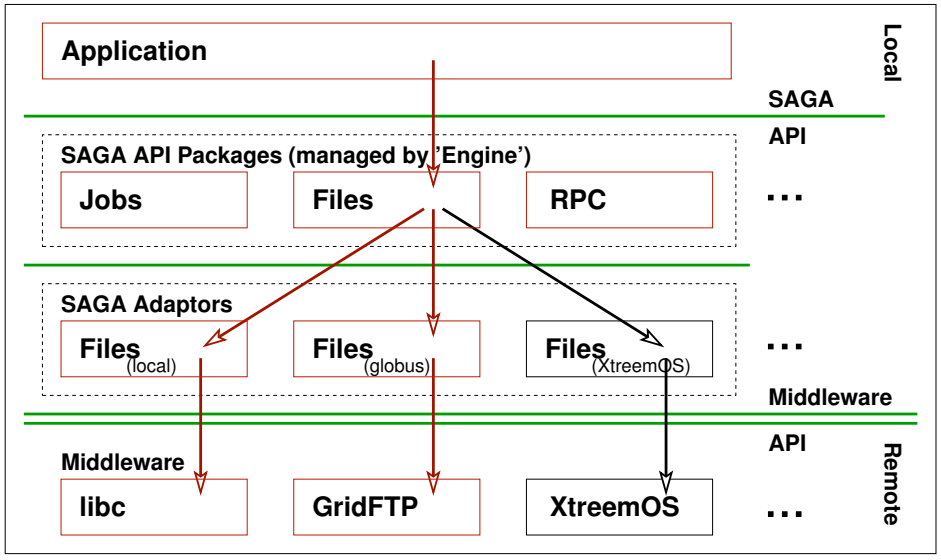

Figure 1: Architecture: A lightweight *engine* dispatches SAGA calls to dynamically loaded middleware *adaptors*.

The SAGA C++ Engine

One of the technically challenging requirements of the SAGA Core API specification is that SAGA object copy operations are shallow copies by default, so copies do not perform a deep copy of object state. These semantics are performant in remote environments as they avoid remote operations (state query and duplication) in most cases.

A second challenge is that the lifetime of a SAGA object is *not* only defined by its scope in the program, but depends (a) on the lifetime of objects depending on that instance, (b) pending asynchronous operations for that instance, and (c) shallow copies of that instance.

To address these challenges, our SAGA implementation uses a technique called the PIMPL mechanism (**p**rivate **impl**ementation), shown in Figure 2. Using this

technique, we were able to simplify the internal state management of SAGA objects and to resolve the lifetime dependencies between SAGA objects, SAGA sessions, and adaptors [4]. At the same time, the engine provides the complete SAGA task model, e.g. implements all SAGA operations asynchronously, even if that is not explicitly supported by the backend services. Both the central call routing and the central management of asynchronous operations, allow for smart runtime optimizations of the remote method invocations [3], which are, for example, exploited for bulk optimizations.

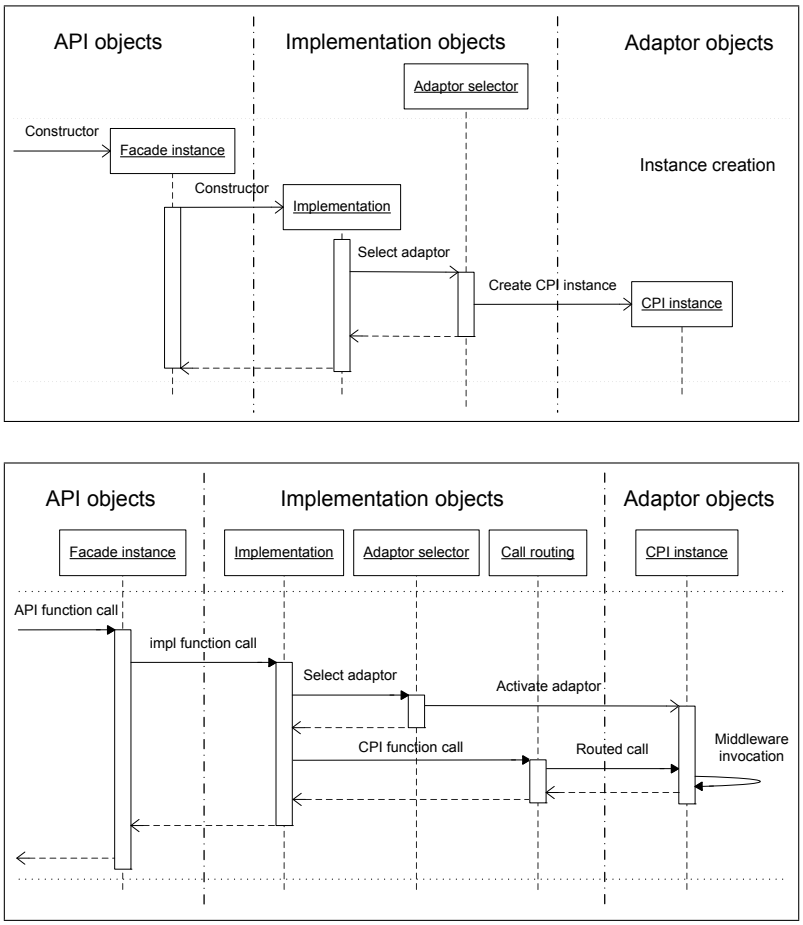

Figure 2: The PIMPL mechanism hides the implementastion from the end user. Shown are object creation (top) and invocation of a SAGA function call (bottom).

Using the PIMPL mechanism, the SAGA object does not maintain any state itself, but is merely a facade maintaining a private, shared pointer to the implementation of the (stateful) SAGA object, and all method invocations are simply forwarded to that implementation instance. On copies, a new facade instance is cre-

ated which maintains another shared copy to the same implementation instance, using, by definition, shallow copy semantics, as the stateful implementation is not copied at all. Also, depending objects and task instances (which represent asynchronous operations) maintain additional shared pointers to the implementation instance and are thus extending the lifetime of that instance: only when all shared pointer copies are finally freed (i.e. when all depending objects are deleted and all asynchronous operations are completed) is the stateful implementation deleted.

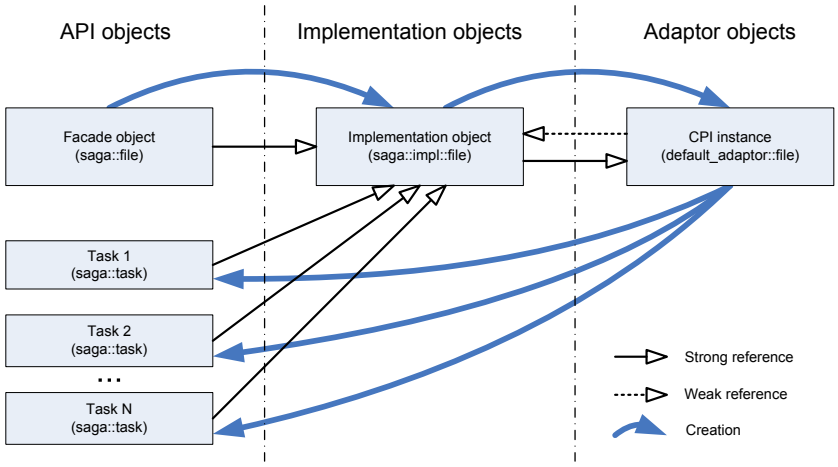

Figure 3: Shared pointers to the implementation object instance define the lifetime of the SAGA objects.

As can be seen in Figure 3, we also use the shared pointer abstraction for the internal lifetime management of the adaptor instances: multiple of those instances can co-exist and provide the implementation (i.e., middleware binding) of the SAGA object implementations.

Summarizing our experiences with implementing the SAGA API, we have learned the following lessons:

- As almost all grid operations are to a remote service, communication latency and middleware invocation overhead dominate local invocation costs. This allows API implementations to apply elaborate (and traditionally expensive) optimizations, and clean and simple (and also traditionally expensive) implementation approaches: a couple of elements in the call stack simply do not matter (as they cost at most micro- or maybe milliseconds), compared to remote communication latency and middleware invocation overhead that may sum up to tens of seconds.

- It is possible and advisable to provide an extensible implementation for an extensible API. This lesson seems self-evident, but needs to be followed in

multiple dimensions: implementation, maintenance, support, documentation, tutorials, optimizations, etc.

- Software complexity does not disappear by introducing new layers – complexity just gets moved to a different layer. In our case, the complexity of interfacing to grid middleware is moved out of the application code (as intended) and into the SAGA adaptors. As such, we consider our implementation approach to be highly successful.

The SAGA C++ Adaptors

Along with the SAGA engine, which is providing the SAGA API itself, we have started to implement the appropriate middleware bindings, i.e. SAGA adaptors. Firstly, local adaptors have been provided which interface to the local operating system (in the case of XtreemOS: Linux) and provide the SAGA functionality on the local host machine, as well as LinuxSSI clusters. Besides, the local adaptor set is also important for (a) development and debugging purposes, and (b) as reference for other, non-local adaptors.

Further, the SAGA implementation includes experimental adaptors to the Globus (GT4) GridFTP service for file access and file management, and to the Globus (GT4) GRAM service for job submission and management. Obviously, the next step is to interface directly with the Xtreemos services, and thus to provide XtreemOS adaptors. This is subject to future deliverables.

4 Installation and deployment of the runtime engine

Our C++ SAGA implementation is an open source project, a collaboration with other parties interested in developing the SAGA API. It is recommended to download the latest stable release, available from the SAGA website. For historical reasons, the SAGA website is located at:

```
http://saga.cct.lsu.edu
```

The development source tree, with the most recent snapshot (not always guaranteed to be free of problems), can be found in the public subversion repository:

```
svn co https://svn.cct.lsu.edu/repos/saga/trunk
```

The SAGA source distribution contains

- a quick introduction (`./README`)

- compile instructions (`./INSTALL`)
- compile environment for Unix/gmake, Unix/Eclipse, Windows/Visual Studio, and MaxOS-X/XCode (`./projects/`)
- several examples and tutorial material (`./examples/`)
- source code of the C++ engine
- a complete set of local adaptors
- Globus job (GRAM) and file (GridFTP) adaptors.

SAGA depends on the free Boost C++ libraries, version 1.33.1 or higher. They can usually be found in the package repository of your Linux distribution. Alternatively, they can be downloaded from `http://www.boost.org`.

Below, the Linux/make installation guidelines (relevant for XtreemOS) are quoted from the `INSTALL` file of the source distribution.

```
----------------------------------------------------------------
Quick Start - UNIX/Linux

Build & Install SAGA
--------------------

  - Configure SAGA: configure
    Run configure in the top directory of the source
    distribution.  You can pass a lot of options to configure.
    At least, you should use option --prefix to specify where
    to install the library.  For example:

      ./configure --prefix=<DESTDIR>

    To see other options of configure run: ./configure --help

    For example, you can pass --without-<PACKAGE> to exclude a
    specific package from the build process.

  - Build SAGA: make
    NOTE: If you don't have a Globus installation, make will
    fail building the Globus Adaptor set. We don't have a
    configure switch to exclude the Globus adaptors from the
    build process yet. Don't bother about it! 'make install'
    will still install all required parts of SAGA.
```

- Install SAGA: make install
  This will create (in case it doesn't exist) a directory
  (--prefix=) containing:

```
lib/           -> shared libraries and adaptors
include/saga/  -> header files
share/saga     -> configuration (.ini) files
```

Build & Install the Globus Adaptor Set

- Set GLOBUS_LOCATION and GLOBUS_FLAVOUR variables according
  to your system settings:

```
export GLOBUS_LOCATION=/usr/local/globus (default)
export GLOBUS_FLAVOUR=gcc32dbgpthr (default)
```

- cd adaptors/globus4-preWS/

- Build the adaptors: make (please ignore compiler warning ;-)

- Install the adaptors make installs

Configure SAGA Environment

To help SAGA to find its configuration files, you should set
the SAGA_LOCATION variable to <DESTDIR>:

```
export SAGA_LOCATION=<DESTDIR>
```

If you have installed SAGA to a non default location on you
system you may also want to add the SAGA libraries to the
dynamic loader's include path:

- On Linux and most other Unix systems:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$SAGA_LOCATION/lib
```

- On MacOS X:

```
    export DYLD_LIBRARY_PATH=$DYLD_LIBRARY_PATH:\
                             $SAGA_LOCATION/lib
```

If you want SAGA to produce more verbose output and debugging statements, you can set the SAGA_VERBOSE variable to a non-zero value:

```
    export SAGA_VERBOSE=1
```

A Simple 5 Minutes SAGA Example

Here's a small program reading the content of a file using SAGA's package_file I/O methods:

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

int main( int argc, char* argv[] )
{
  if ( argc < 2 )
  {
    std::cout << "\nUsage: " << argv[0]
                             << " [URL] \n"
                             << std::endl;
  }
  else
  {
    saga::size_t readbytes = 0;
    char inbuff[64];

    try
    {
      saga::file f (argv[1], saga::file::Read);

      while ( readbytes = f.read (saga::buffer (inbuff)) )
      {
        std::cout << std::string(inbuff,readbytes)
```

```
                  << std::flush;
      }
    }
    catch (saga::exception const &e)
    {
      std::cout << "Couldn't read the file: "
                << e.what()
                << std::endl;
    }
  }
  return 0;
}
//
////////////////////////////////////////////////////////////////
```

Copy the source to a file "saga_file_read.cpp" and compile it using the following command:

```
g++ saga_file_read.cpp -o saga_file_read               \
    -I$SAGA_LOCATION/include                           \
    -I$BOOST_LOCATION/include -L$SAGA_LOCATION/lib/    \
    -lsaga_engine -lsaga_package_namespace             \
    -lsaga_package_file
```

The resulting binary should be able to read and print out the contents of arbitrary local files on your system - for example:

```
./saga_file_read any://localhost/.bashrc
```

should print out the content of your .bashrc file.

NOTE: If you try to substitute localhost with a hostname that doesn't point to your local machine, the execution will fail if you haven't built the Globus GridFTP adaptors. Otherwise it should work on Globus resources as long as you have a valid Grid Proxy Credential.

Have fun!

The SAGA development team.

---

5 API documentation

API documentation is available in three different formats. Firstly, the OGF SAGA API standard document [2] is, naturally, a comprehensive documentation source for the SAGA API. Secondly, a number of tutorials are included in the released code package. And finally, a detailed API documentation is generated by doxygen. It is available from

```
http://saga.cct.lsu.edu/apidoc/
```

6 Summary and Future Work

In this report, we have presented the first prototype of the XtreemOS runtime engine, implementing the SAGA API, according to our previous deliverable D3.1.1. We have outlined the underlying design principles of our implementation, and have provided information for download, installation, and use.

In a companion deliverable, D3.1.2, we are presenting the second draft API for XtreemOS, focusing on SAGA extensions for exposing XtreemOS-specific functionality. This functionality will be implemented in later releases. Other directions of future work are language bindings to the SAGA API. The current implementation defines an ad-hoc language binding to C++, which still has to undergo the standardization process within OGF. Other required language bindings (like Java, C, Fortran) need to be developed later, as well.

References

[1] Gabrielle Allen, Kelly Davis, Tom Goodale, Andrei Hutanu, Hartmut Kaiser, Thilo Kielmann, Andre Merzky, Rob van Nieuwpoort, Alexander Reinefeld, Florian Schintke, Thorsten Schütt, Ed Seidel, and Brygg Ullmer. The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid. *Proceedings of the IEEE*, 93(3):534–550, 2005.

[2] Tom Goodale, Shantenu Jha, Hartmut Kaiser, Thilo Kielmann, Pascal Kleijer, Andre Merzky, John Shalf, and Christopher Smith. A Simple API for Grid

Applications (SAGA). Grid Forum Document GFD.90, 2007. Open Grid Forum (OGF).

[3] Stephan Hirmer, Hartmut Kaiser, Andre Merzky, Andrei Hutanu, and Gabrielle Allen. Generic Support for Bulk Operations in Grid Applications. In *MCG '06: Proceedings of the 4th International Workshop on Middleware for Grid Computing*, page 9, New York, NY, USA, November 2006. ACM Press.

[4] Hartmut Kaiser, Andre Merzky, Stephan Hirmer, and Gabrielle Allen. The SAGA C++ Reference Implementation – Lessons Learnt from Juggling with Seemingly Contradictory Goals. In *Workshop on Library-Centric Software Design LCSD'06, at Object-Oriented Programming, Systems, Languages and Applications conference (OOPSLA'06)*, Portland, Oregon, USA, October 2006.

[5] Andreas Savva (Ed.). JSDL SPMD Application Extension, Version 1.0. Grid Forum Document GFD.115, 2007. Open Grid Forum (OGF).

[6] First Draft Specification of Programming Interfaces. Deliverable D3.1.1, XtreemOS consortium, 2006.

[7] Second Draft Specification of Programming Interfaces. Deliverable D3.1.2, XtreemOS consortium, 2007.