SAGA C++

Ole Weidner, CCT/LSU
Hartmut Kaiser, CCT/LSU

Version: 0.3

September 2, 2009

SAGA C++ Installation Manual

Abstract

This document describes the installation and configuration process for the SAGA C++ Core Libraries and Adaptors. This document should always reflect the latest changes and additions made to the build and installation system. However, please make sure to read the `README` file provided with each SAGA distribution for last minute changes and informations.

Please help us to improve the quality of SAGA and file a bug report if you have encountered any problems with the build system. Our bug-tracking system can be found at:

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

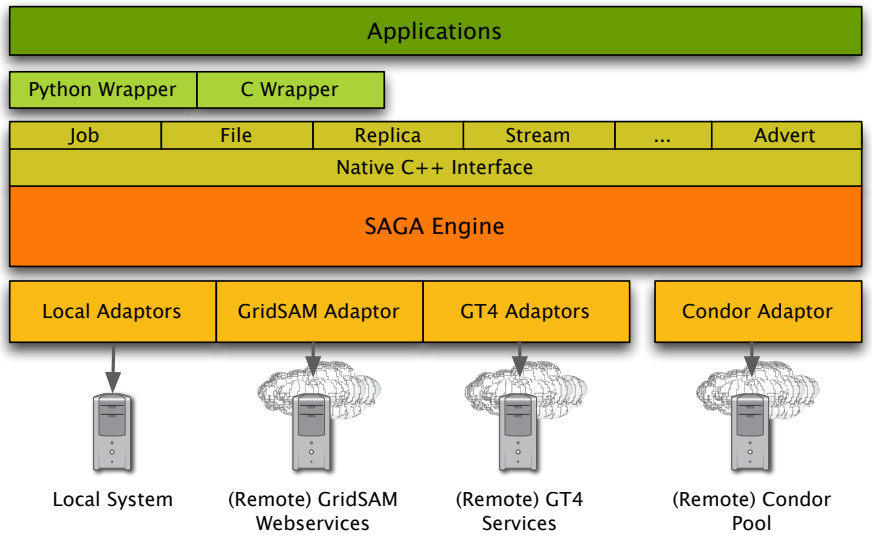
Contents

1 Overview

SAGA consists of three different component types: (1) the *Engine* with its *Packages* and the native *C++ Interface*, (2) the *Language Bindings* (Python, C), and (3) the *Adaptors* (Local, GT4, ...). All components are part of the source distribution. The *Engine* provides the core functionality which leads to the following dependencies:

- Language Bindings require the SAGA Engine
- Adaptors require the SAGA Engine

This means that it is always required to build and install the SAGA Engine. It is, for example not possible to build and install just the Python language bindings without having the SAGA Engine built before. On the other hand, just building the Engine without any middleware Adaptors is quite useless ;-)



The SAGA build system takes this into account and configures and builds all SAGA components which are supported on the target system in the right order. However, it is possible to configure and build each component individually. The following sections describe both, how to build all components in one batch as well as how to build each component individually.

2 Platform Notes

One of the major design goals of the SAGA C++ implementation was to create platform-independent and portable API which is crucial in heterogeneous distributed environments like Computational Grids. Although it is possible to build SAGA on any operating system there are limitations and particularities on certain platforms. Furthermore, some of the SAGA Adaptors require 3rd party libraries that may not be available on all supported platforms.

2.1 Microsoft Windows

You can build SAGA on Microsoft Windows using Microsoft's Visual C/C++ compiler. However, the `make`-based build system described in Section 4 will not work on Windows unless you install the *Cygwin* environment. The easiest way to build SAGA on Windows is using the IDE project file for *Microsoft Visual Studio*.

2.2 IBM AIX 5L

SAGA supports both, the GNU `gcc` as well as the IBM XlC compiler, although we strongly suggest to use `gcc`. Building 64bit version of SAGA doesn't work with any of the compilers due to some yet unsolved linkage issue.

The `make` implementation that comes with AIX does NOT work with the SAGA build-system. Please install and use GNU `make`. It's part of the free *AIX Toolbox for Linux Applications* which can be found here: http://www-03.ibm.com/systems/p/os/aix/linux/index.html

3 Requirements

In order to build the different SAGA components from source, a couple of external libraries are required. We've tried to keep the requirement for the *Engine* down to a bare minimum, but some of the middleware *Adaptors* and *Language Bindings* require additional external libraries and tools.

3.1 Compilers

Although the SAGA build-system should support all major C++ compilers, we highly recommend the use of the GNU `gcc` which should be available on any platform. The following table gives an overview of compilers and versions that are known to work with SAGA:

Compiler	Minimum Version	Notes
GNU g++	$\geq$ 3.4.6	-
IBM xlC++	$\geq$ 8.0	-
MS Visual C++	$\geq$ 7.0	-

3.2 Engine Requirements

The SAGA Engine makes extensive use of the Open Source Boost C++ Libararies, namely the Boost `thread`, `iostreams`, `serialization`, `filesystem` and `regex` packages. Having Boost installed is the only **mandatory** requirement for building any of the SAGA components.

Requirements	Version	Notes
Boost C++ Libraries	$\geq$ 1.34.1	http://www.boost.org

3.3 Language Binding Requirements

Currently, we provide language bindings for C and Python. The C language binding doesn't need any external requirements except a working C/C++ compiler. The Python language binding requires a recent version of Python and

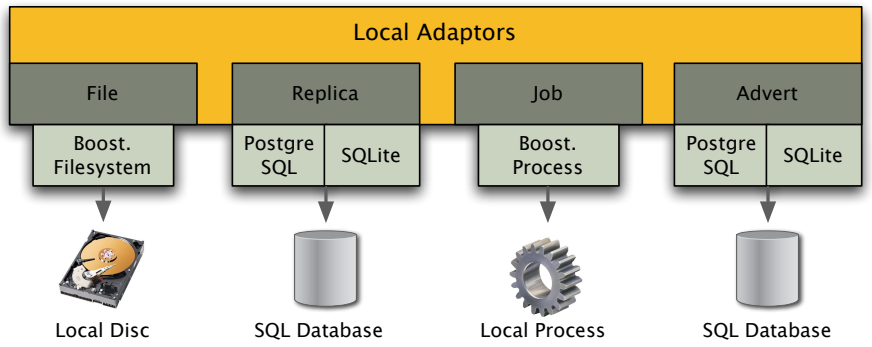
the Python interoperability libraries that come with Boost. Since the Python bindings require decent threading support, we require at least Python 2.4.

NOTE: If you're building your own Python libraries, make sure that build them using the `--enable-shared` flag **before** you build Boost or SAGA.

Requirements	Version	Notes
Boost C++ Libraries	$\geq$ 1.34.1	all
Python	$\geq$ 2.4	http://www.python.org

3.4 Adaptor Requirements

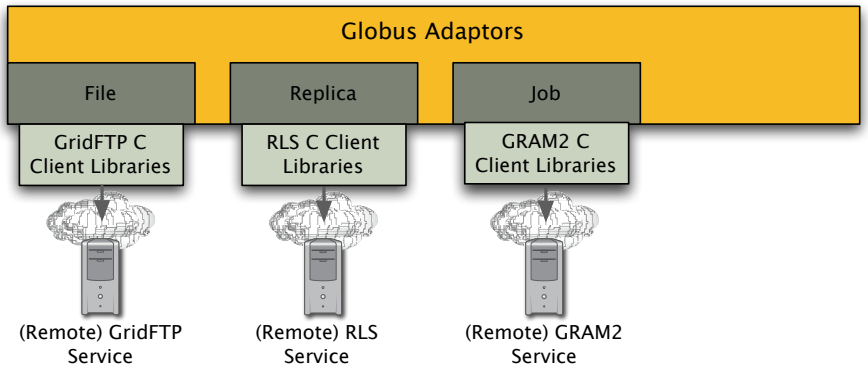
3.4.1 Local Adaptors



The local adaptor set uses the Boost.`filesytem` to access local filesystems and Boost.`process` to spawn and control local processes (jobs). The Replica and Advert adaptors use the PostgreSQL *or* SQLite3 client libraries to acces local or remote PostgreSQL or SQLite databases. Only one of the libraries is required to build the adaptors. However, if both are found, the adaptors gets built with support for both databases.

Requirements	Version	Package(s)
Boost C++ Libraries	$\geq$ 1.34.1	all
SQLite3	$\geq$ 3.3	Advert & Replica
PostgreSQL	$\geq$ 8.0	Advert & Replica

3.4.2 Globus Toolkit Adaptors



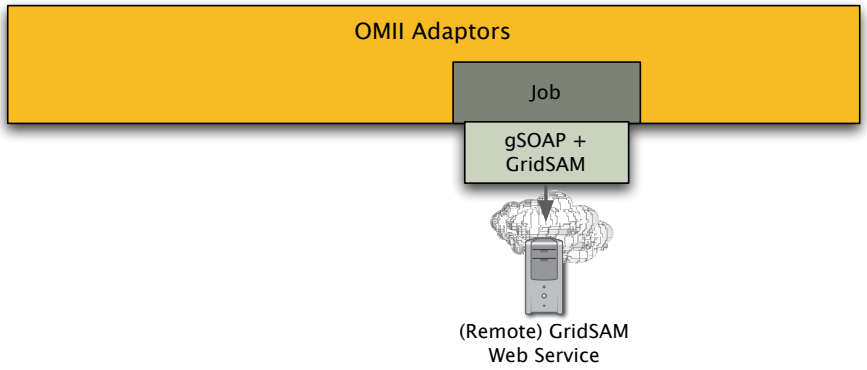
In order to build the Globus Toolkit adaptors, you need a **local installation** of the Globus **C header and client library** files. It is **not required** to have any of the local Globus services configured or running. The adaptors use the Globus GridFTP, GRAM2, RLS and their dependent (XIO, etc.) client libraries. Note that the **developer packages** for Globus need to be installed, in order to have the header files available. Alternatively, Globus can be installed from source – that will also make the Globus header files available.

Requirements	Version	Package(s)
Boost C++ Libraries	$\geq$ 1.34.1	all
Globus Toolkit	$\geq$ 3.2.1	all

3.4.3 OMII GridSAM Adaptor

The OMII GridSAM adaptor uses the SOAP protocol to communicate with a local or remote GridSAM Web-Service.

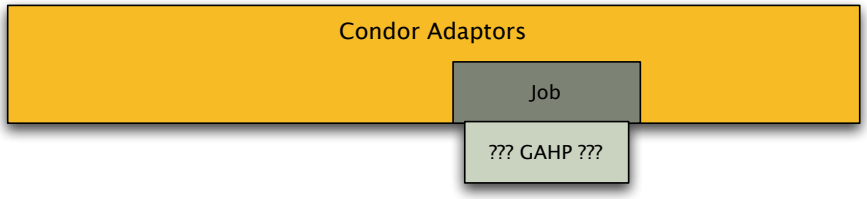
The adaptor comes with its own version of gSOAP and pre-generated WSDL stubs. This means that the adaptor doesn't have any external requirements



except (Open)SSL and libCrypt (for HTTPS/encryption support) which should be available on any system.

Requirements	Version	Package(s)
Boost C++ Libraries	$\geq$ 1.34.1	all
(Open)SSL	$\geq$ any ??	all
libCrypt	$\geq$ any ??	all

3.4.4 Condor Adaptor



The Condor adaptor communicates with the Condor command line tools – thus, no client library is required, but a working a configured condor client installation is expected to be available. The environment `CONDOR_LOCATION` is evaluated by configure to find that location. The variable typically used by condor, `CONDOR_CONFIG`, is also evaluated when `CONDOR_LOCATION` is not present.

Requirements	Version	Package(s)
Boost C++ Libraries	$\geq$ 1.34.1	all
Condor client installation	$\geq$ any ??	all

4 Installing SAGA from Source

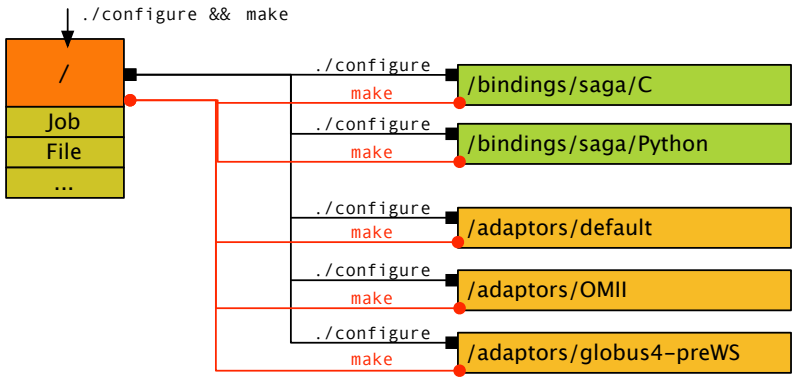
Now that you have checked the prerequisistes for the different SAGA packages in section 3, you can go on and build SAGA from source.

4.1 Getting the Sources

The SAGA sources are available from http://sourceforge.net/projects/saga in gzip, bzip2 and zip format. All archives contain exactly the same files.

4.2 Build System Structure

The SAGA build-system is based on autoconf and configure. We require the use of GNU `make` to build SAGA. Please also check the *Platform Notes* (section 2) for known issues with default make installations (e.g. on AIX).



The SAGA source-tree provides seperate build-systems for the Engine & Packages, for the different Language Bindings, and for the Adaptors. However, the top-level `./configure` and Makefiles try to recursively configure and (if successful) build all available SAGA components.

If you want to configure and build any component separately, you can call ./configure and make directly in the component's subdirectory.

4.3 Building SAGA - The Simple Way

This section assumes that you have installed all prerequesites for the components you want to install. In case something is missing, the build-system will NOT necessarily fail but rather skip the component.

4.3.1 Configure All Components

As described in section 4.2, the top-level configure script recursively calls the configure scripts for all SAGA components:

```
> ./configure --prefix=/saga/install/path
```

or to be consistent:

```
> ./configure --prefix=$SAGA_LOCATION
```

The build-system tries to find required libraries like Boost in the system's default installation locations (e.g. `/usr`, `/usr/local`) on Linux). In case you have installed them somewhere else, you have to provide ./configure with the appropriate paths using the following arguments:

- `--with-boost=DIR Path to Boost installation`

There are lots of other parameters and environment variables that will affect the behaviour of `configure`. For a complete overview, type:

```
> ./configure --help
```

While configure runs, you will see the configuration summaries for each SAGA component. The summary will tell you if all requirements are met in order to build the component and if not, which requirements are missing. Here is an example output of the Python bindings summary:

```
configure: ========================================================
configure: SAGA Python BINDINGS - Configuration Summary
configure: ========================================================
configure:
configure: SAGA Source           : /tmp/trunk
configure: SAGA Location         :
configure: Install Prefix        : /usr/local
configure: Python Package Path   : lib/python2.5/site-packages/saga
configure:
configure: Using SAGA from       : /tmp/trunk (source)
configure:
configure: Python Found          : yes
configure: Python Version        : 2.5
configure: Python Location       : /usr
```

4.3.2 Build and Install all Components

After you have successfully configured the sources, it's time to build them. Similar to the top-level configure, the top-level Makefile will recursively build all configured components:

```
> make && make install
```

Once `make install` is done, you will find your freshly built SAGA in the directory you provided with `--prefix`.

On multiprocessor or multicore machines, SAGA can be build with the `-j<n>` option, with `<n>` specifying the maximal number of parallel build processes.

4.3.3 Setup the Environment

The SAGA Engine needs to know where to look for its configuration files in order to configure and load the middleware adaptors. By default, the installation prefix is used to load these configuration files. The easiest way to tell the Engine where to find alternative settings is by setting the `$SAGA_LOCATION` variable (e.g. in your .bash_profile) to a SAGA installation directory (set via the `--prefix` option during configure). More details on SAGA runtime configurations can be found in section 4.4.

In case the installation directory is not one of the standard directories (`/usr/` and `/usr/local` on Linux), you should set the `$LD_LIBRARY_PATH` (on Mac OS: `$DYLD_LIBRARY_PATH`) as well:

```
> export SAGA_LOCATION=/saga/install/path
> export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$SAGA_LOCATION/lib
```

4.3.4 Setup the Python Bindings

If you want to use the Python bindings, you have to add the installation directory of the language bindings to your `$PYTHONPATH`:

```
> export PYTHONPATH=$PYTHONPATH:$SAGA_LOCATION/lib/python2.5/site-packages/
```

You can test the Python bindings using the Python interpreter from the commandline. The following set of commands should display the interface definition for the SAGA file package:

```
> python
Python 2.5.1 (r251:54863, Jan 17 2008, 19:35:17)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> import saga

>>> help(saga.file)
```

4.3.5 Running the Tests

NOTE: Make sure you have configured your `$SAGA_LOCATION` and your library path as described in section 4.3.3 - otherwise the test won't run!

SAGA comes with a more or less complete set of unit tests for the packages, the engine and for some of the adaptors. You can run them directly from the source root using the build-system command:

```
> make check
```

Please help us to improve the quality of SAGA and file a bug report if you have encountered any problems with the tests. Our bug-tracking system can be found at: `http://saga.cct.lsu.edu/cpp/dev/`

4.4 Configuring the SAGA Engine

The SAGA engine module is responsible for dispatching any API call executed by an application to a corresponding CPI function (CPI: capability provider interface). The CPI functions are implemented by adaptors, generally, one adaptor for each different (grid) service.

The selection of an appropriate adaptor is done based on a capability registry maintained by the SAGA engine. This registry contains entries for all CPI functions implemented by all adaptors known to the engine. The registry is filled at startup by the adaptors, generating the corresponding CPI function descriptors and registering these with the engine.

The SAGA engine uses a system of configuration files to discover the adaptors to load. Each execution environment should provide at least one main configuration file (most of the time called `saga.ini`) containing a list of directories to scan for the adaptor specific configurations files. This main configuration file should specify at least one directory, specifying where to look for these adaptor configuration files (ini files):

```
[info]
  ini_path = ${SAGA_LOCATION}/share/saga/
```

4.4.1 Ini File Syntax

Ini files are structured by *sections*, such as `[saga]`, `[saga.adaptors]`, `[saga.adaptors.default]` etc. In these sections, which form a hierarchy, keys can be assigned arbitrary values

```
[section.subsection]
  key = value
```

For any value (right hand side of an `'='`) in any of the configuration files the notation `${SOME_ENV}` allows to use values currently defined by `SOME_ENV` environment variable. Similarly, `$[section.key]` can be used to refer to an ini variable defined elsewhere. Also, a fallback can be specified if some key is not available

```
[section.subsection]
  shell   = ${SHELL}
```

```
  check   = $[section.subsection.shell]
  verbose = ${SAGA_VERBOSE:info} # use 'info' as fallback
```

4.4.2 SAGA configuration files

First of all: if you installed a SAGA engine and a set of adaptors from sources, or from a packaged binary, everything should be configured so that no additional configuration steps are required. The following guidelines are for those cases, where (a) a system administrator wants to configure SAGA for a specific invironment, or (b) a SAGA user wishes or needs to overwrite the default or system settings.

The SAGA engine searches for the main configuration file at the following locations:

- A file `/etc/saga.ini` (non Windows platforms only)
- A file `$SAGA_LOCATION/share/saga/saga.ini`
- A file `$HOME/.saga.ini`
- A file `$PWD/.saga.ini`
- A file $SAGA_INI as defined in the environment

All found `info.ini_path` keys in these files are concatinated and the corresponding directories are scanned for adaptor configuration files in the order as they appeared. The `info.ini_path` key may contain a list of directories, separated by colons (on Windows the separator character is a semicolon). Any file having the file extension `.ini` in one of the listed directories will be treated as an adaptor configuration file.

Each of the adaptor configuration files can contain the following entries:

```
[saga.adaptors.<adaptor_name>]
  name    = <adaptor_instance_name>
  path    = <adaptor_path>
  enabled = true
```

where

- `<adaptor_name>` should be replaced by some adaptor specific name used to identify the adaptor module.

- `<adaptor_instance_name>` should be replaced by some unique name identifying the adaptor instance to be loaded
- `<adaptor_path>` is either the full path of the adaptor module (shared library) to load or it should point to the directory, where the adaptor module is located. In the later case the name of the adaptor module should conform to the naming convention: `libsaga_adaptor_<adaptor_name>.so` (on different operating systems the file extension might vary).
- `enabled` is a bool which determines if the adaptor is considered for loading. Valid values are `true` and `false`.

All keys apart from `name` are optional, and have fallbacks consistent with the original installation configuration.

As an example, the simplest possible adaptor configuration file for a `default_file` adaptor might look as follows:

```
saga.adaptors.default_file]
  name = default_file
```

Below, the default saga.ini is listed completely.

```
aga]
# saga install root,  points to what is set in the environment, as
# SAGA_LOCATION, or use the configure time prefix as fallback.
location = ${SAGA_LOCATION:/Users/merzky/links/saga/install/trunk}

# where to find adaptor ini files
ini_path = $[saga.location]/share/saga/
```

All adaptor and user ini files should refer to `$[saga.location]` when refering to the saga installation root, as that evaluates `$SAGA_LOCATION`, and provides a sensble fallback, the installation prefix. The installation prefix is also hardcoded in the `libsaga_engine`, so the engine should be able to find `saga.ini`, and thus the adaptor inis, even when `SAGA_LOCATION` is not set in the environment.

4.4.3 SAGA Lite

By default, the above procedure creates a set of shared libraries, one for the SAGA engine (`libsaga_engine`), and a number of SAGA packages (`libsaga_package_abc`). Also, the adaptor libraries are created as shared libs (`libsaga_adaptor_xyz`).

Additionally, a `libsaga_lite` library is created, which combines *all* of the above libraries into a single library, thus simplifying deployment, application linkage, and runtime setup for SAGA (more details, see programming manual). The set of adaptors compiled into the Lite version is determined by the file `dynamic_adaptor.list` in the `lite/` subdirectory. That file is created by running `make` there. For changing that list, run make once, then edit that file, and run `make` again to create the correct library setup[1].

[1] That procedure will be simplified in the future.