

# The SAGA C++ Reference Implementation

Lessons Learnt from Juggling with Seemingly Contradictory Goals

Hartmut Kaiser, Andre Merzky, Stephan Hirmer,  
Gabrielle Allen





# CCT Roadmap

- What is SAGA, why do we think, we need it?
  - Why another Grid API
  - Realities of Grids
  - Simple API for grid Applications
- Requirements
- Architecture
- Lessons learnt
- Code examples



# Simple API for Grid Applications

- OGF standardization effort
  - Intends to simplify the development of grid-enabled applications by standardizing syntax and semantics of a Grid API
  - Even for scientists with no background in computer science, or grid computing
  - Interesting, challenging problem due to dynamic environment of today's Grids
  - 80/20 rule: maximal abstraction for minimal programming effort for the user
- *Simple* and *Standard* are the keywords



# Why another Grid API?

The situation today:

- Grids: everywhere
  - » Well, supposedly. At least many projects ☺
- Grid applications: nowhere
  - » Almost. At least our in experience that this is difficult

Why is this?

- Application programmers accept the Grid as a computing paradigm only very slowly.
- Problems: (multifold and often cited - amongst others)
  - Interfaces are not simple (see next slides. . .)
    - » Typical Globus code... ahem... ☺
  - Different and evolving interfaces to the 'Grid'
    - » Versions, new services, new implementations, WSDL or WSRF do not solve all problems at all
  - Environment changes in many ways
    - » Globus, grid members, services, network, applications, ...



# Copy a File: Globus GASS

```
int copy_file (char const* source,   char const* target)
{
    globus_url_t                source_url;
    globus_io_handle_t          dest_io_handle;
    globus_ftp_client_operationattr_t source_ftp_attr;
    globus_result_t              result;
    globus_gass_transfer_requestattr_t source_gass_attr;
    globus_gass_copy_attr_t      source_gass_copy_attr;
    globus_gass_copy_handle_t     gass_copy_handle;
    globus_gass_copy_handleattr_t gass_copy_handleattr;
    globus_ftp_client_handleattr_t ftp_handleattr;
    globus_io_attr_t             io_attr;
    int                          output_file = -1;

    if ( globus_url_parse (source_URL, &source_url) != GLOBUS_SUCCESS ) {
        printf ("can not parse source_URL \"%s\"\n", source_URL);
        return (-1);
    }

    if ( source_url.scheme_type != GLOBUS_URL_SCHEME_GSIFTP &&
        source_url.scheme_type != GLOBUS_URL_SCHEME_FTP    &&
        source_url.scheme_type != GLOBUS_URL_SCHEME_HTTP   &&
        source_url.scheme_type != GLOBUS_URL_SCHEME_HTTPS  ) {
        printf ("can not copy from %s - wrong prot\n", source_URL);
        return (-1);
    }

    globus_gass_copy_handleattr_init (&gass_copy_handleattr);
    globus_gass_copy_attr_init       (&source_gass_copy_attr);

    globus_ftp_client_handleattr_init (&ftp_handleattr);
    globus_io_fileattr_init           (&io_attr);

    globus_gass_copy_attr_set_io      (&source_gass_copy_attr, &io_attr);
    globus_gass_copy_handleattr_set_ftp_attr
        (&gass_copy_handleattr,
         &ftp_handleattr);

    globus_gass_copy_handle_init      (&gass_copy_handle,
        &gass_copy_handleattr);
```

```
    if (source_url.scheme_type == GLOBUS_URL_SCHEME_GSIFTP ||
        source_url.scheme_type == GLOBUS_URL_SCHEME_FTP    ) {
        globus_ftp_client_operationattr_init (&source_ftp_attr);
        globus_gass_copy_attr_set_ftp (&source_gass_copy_attr,
                                         &source_ftp_attr);
    }
    else {
        globus_gass_transfer_requestattr_init (&source_gass_attr,
                                                source_url.scheme);
        globus_gass_copy_attr_set_gass (&source_gass_copy_attr,
                                         &source_gass_attr);
    }

    output_file = globus_libc_open ((char*) target,
        O_WRONLY | O_TRUNC | O_CREAT,
        S_IRUSR | S_IWUSR | S_IRGRP |
        S_IWGRP);

    if ( output_file == -1 ) {
        printf ("could not open the file \"%s\"\n", target);
        return (-1);
    }

    /* convert stdout to be a globus_io_handle */
    if ( globus_io_file_posix_convert (output_file, 0,
        &dest_io_handle)

        != GLOBUS_SUCCESS) {
        printf ("Error converting the file handle\n");
        return (-1);
    }

    result = globus_gass_copy_register_url_to_handle (
        &gass_copy_handle, (char*)source_URL,
        &source_gass_copy_attr, &dest_io_handle,
        my_callback, NULL);

    if ( result != GLOBUS_SUCCESS ) {
        printf ("error: %s\n", globus_object_printable_to_string
            (globus_error_get (result)));
        return (-1);
    }

    globus_url_destroy (&source_url);
    return (0);
}
```



# Copy a File: SAGA

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

void copy_file(std::string source_url, std::string target_url)
{
    try {
        saga::file f(source_url);
        f.copy(target_url);
    }
    catch (saga::exception const &e) {
        std::cerr << e.what() << std::endl;
    }
}
```

- Provides a standardized high level abstraction layer
- Uses well known paradigms
- Shields programmers from idiosyncracies of Grids
- Blends well with existing language features and libraries



# Realities of Grids

- Different hardware
  - Chipset
  - Architecture (desktops, beowulf clusters, SMP, etc.)
- Different software
  - Operating system, compilers
  - Libraries, software stack
  - Middleware service versions
- Different administrative policies
  - Access policies, quotas, upgrade policies
- Highly dynamic at all levels
  - without notifying you as the user



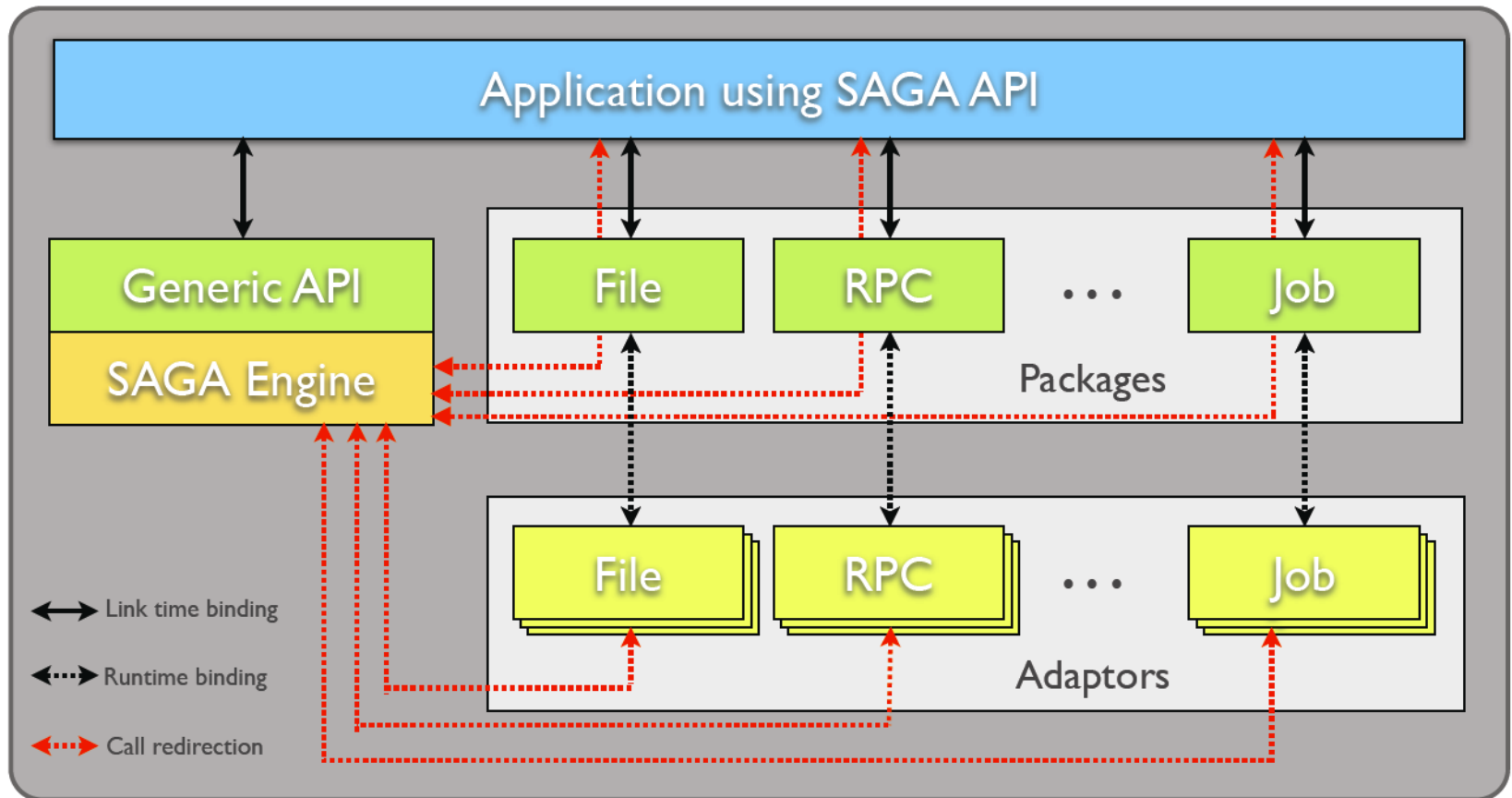
# Requirements

- Provide 'Simple' and 'Easy-to-Use' API
- Cope with evolving Standards landscape and future extensions
  - » OGF Standards are emerging, 5-10 years timeframe until stabilized
- Shield from middleware evolution
  - » Research projects, bad backwards compatibility
- Provide fail safety and hide dynamics
  - » Heterogenous Grid environments
- Portability, platform independence
  - » Distributed applications
- Easy to deploy, configure, and maintain
  - » No. 1 end user requirements
- Provide consistent language bindings
  - » No templates, as close to the spec as possible

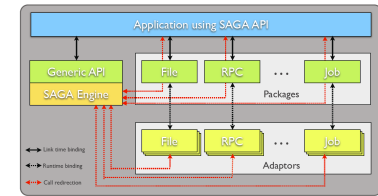




# Architecture Overview



# Extensibility



- Horizontal Extensibility – API Packages
  - Current packages:
    - file management, job management, remote procedure calls, replica management, data streaming, etc.
- Vertical Extensibility – Middleware Bindings
  - Different adaptors for different middleware
  - Set of ‘local’ adaptors
- Extensibility for Optimization and Features
  - Bulk optimization, modular design



# Lessons learnt

- Macros are not (always) evil
- PImpl idiom useful even in complex object hierarchies
- Futures/tasks make it easy to implement asynchronous API's
- Polymorphism is your friend
- Grid environments allow for fancy implementations

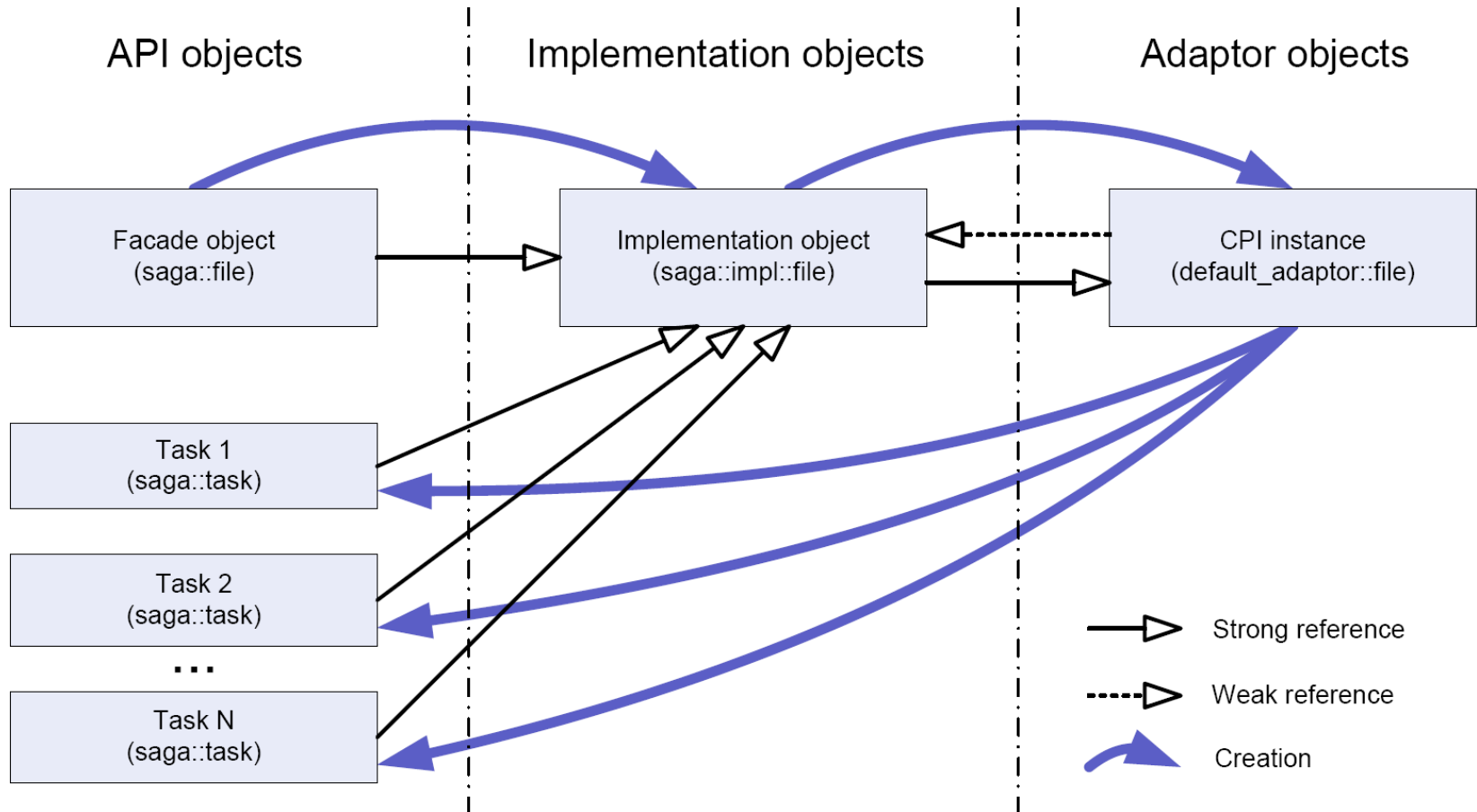


# Macros are not (always) evil

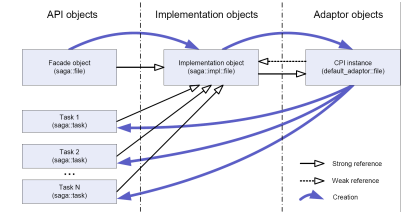
- Every API function in 4 different flavors:  
2 synchronous, 2 asynchronous
  - » Error prone, boring, makes writing new packages a nightmare
- Macros expand to set of functions
  - Type safety not sacrificed
- Partial macro expansion (Boost.Wave)
  - Well, generating partial output
  - No debug problems
- Simplifies writing of new packages



# PImpl idiom



# PImpl idiom



- Simple – not only the API but usage as well
  - Lightweight objects, well blending with existing libraries (STL, Boost, etc.)
  - Avoid object slicing while providing required object hierarchy
  - No need to worry about object lifetimes or memory management
- Decouple API and implementation
- Allow straightforward implementation of asynchronous operations (tasks)
- Tasks are implemented on top of futures
  - Very nice concept, enables simple implementation of fully concurrent and asynchronous API, ensures correct exception handling



# Polymorphism

- Combination of run time and compile time polymorphism
  - I.e. virtual functions (abstract base classes) and templates (code reuse, meta-programming)
  - Run time polymorphism is used to abstract different implementations
  - Compile time polymorphism is used to avoid same code over and over again



# Fancy implementations

- Latencies are major contributor to wall time needed to execute Grid related operations
- Fancy implementations, providing
  - Fallback solutions
  - Sophisticated error handling and fail proof implementations
- These are not only necessary in Grid related scenarios, but also possible





# Example: Task model

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

// normal sync version of the copy method
f.copy ("any://host.net//data/dest1.dat");

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

// task states of the returned saga::task
// t1 is in 'Finished' or 'Failed' state
// t2 is in 'Running' state
// t3 is in 'New' state

t3.run ();

t2.wait ();
t3.wait ();
```



# Example: Bulk optimization

```
std::vector<std::string> files = ...initialize with list of url's...;
saga::task_container tc;

// create file copy tasks
for (std::vector<std::string>::size_type i = 0; i < files.size(); ++i)
{
    saga::file f (files[i]);
    tc.add (f.copy<saga::Task>(...destination url...));
}

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



# Conclusions

- SAGA is a very high abstraction level of Grids using well known programming paradigms
- Libraries are a means of securing investment into the future
- Presented a generic framework for building libraries allowing runtime dispatching to different implementations of similar functionalities