OpenGridForum

Open Forum | Open Standards

SAGA Tutorial - NeSC 2009

Introduction to the SAGA API

Agenda

- Background Recap
- SAGA API structure and scope
- walkthrough
- some implementation details
- SAGA extensions

Grid APIs and Frameworks

- Middleware often targets legacy applications (Unicore, Globus, Condor, ...)
- some are distribution aware (MPICH-G, Ninf-G, . . .)
- few **A**PIs exist for Grid aware applications
  - GridFTP
  - GRAM
  - gLite
  - CoG
  - GAT
  - Cloud APIs

Grid APIs and Frameworks

- diversity of Grid Middleware implies diversity of APIs
- some APIs try to generalize Grid programming concepts
- difficult to keep up with MW development, and to stay **simple**

OGF

- Open Grid Forum (GF, EGF, GGF) tries to standardize Grid MW
- e.g. single job description language (JSDL)
- focuses on interfaces, but also protocol and architecture

APIs within OGF

- OGF focuses on services
- some effort on higher level APIs
  - Distributed Resource Management Application API (DRMAA)
  - Remote Procedure Calls (GridRPC)
  - Checkpoint and Recovery (GridCPR)
  - Job Submission and Description Language (JSDL)
- numerous service interfaces, often WS-based (WSRF)

OGF: DRMAA

- implementable on all major resource management services
- simple means to define jobs, and to submit them
- basic job management features (status, kill)
- job templates for bulk job management

DRMAA Example

DRMAA Job Submit

```
drmaa_job_template_t * job_template;

if ( ! ( job_template = create_job_template (exe, 5, 0) ) )
{
  fprintf (stderr, "create_job_template failed\n");
  return 1;
}

while ( ( drmaa_errno = drmaa_run_job (job_id,
                                       sizeof (jobid)-1,
                                       job_template,
                                       diagnosis,
                                       sizeof (diagnosis)-1)
        ) == DRMAA_ERRNO_DRM_COMMUNICATION_FAILURE )
{
  fprintf(stderr, "drmaa_run_job failed: %s\n", diagnosis);
  sleep (1);
}
```

OGF: GridRPC

- 'standardizes' the three existing RPC implementations for Grids
- example of *'gridified API'*
- simple: get function handle, call function
- explicit support for async rpc calls

OGF: GridRPC

GridRPC: Matrix Multiplication

```
double A[N*N], B[N*N], C[N*N];

initMatA (N, A);
initMatB (N, B);

grpc_initialize (argv[1]);

grpc_function_handle_t handle;
grpc_function_handle_default (&handle, "mat_mult");

if ( grpc_call (&handle, N, A, B, C) != GRPC_NO_ERROR)
{
  exit (1);
}

grpc_function_handle_destruct (&handle);
grpc_finalize ();
```

OGF: GridCPR

- Grids seem to favour application level checkpointing
- GridCPR allows to manage checkpoints
- defines an architecture, service interfaces, and, aehem, no API

OGF: JSDL

- extensible XML based language for describing job requirements
- does not cover resource description (on purpose)
- does not cover workflows, or job dependencies etc (on purpose)
- JSDL is extensible (ParameterSweep, SPMD)

OGF: JSDL

JSDL: Simple Job

```
<jsdl:JobDefinition>
  <JobDescription>
     <Application>
        <jsdl-posix:POSIXApplication>
           <Executable>/bin/date</Executable>
        </jsdl-posix:POSIXApplication>
     </Application>
     <Resources ...>
        <OperatingSystem>
           <OperatingSystemType>
              <OperatingSystemName>LINUX</OperatingSystemName>
           </OperatingSystemType>
        </OperatingSystem>
     </Resources>
  </JobDescription>
<jsdl:JobDefinition>
```

OGF: JSDL

- XML: embeddable into WSRF (WS-Agreement etc.)
- XML, but relatively flat
- maps well to existing JDLs, but is ’more complete’
- extensible (resource description, job dependencies, workflow)
- top down approach!

OGF: Summary

- some APIs exist in OGF, and are successfull
- OGF APIs do not cover the complete OGF scope
- the various API standards are disjunct
- WSDL as service interface specification cannot replace an application level API (wrong level of abstraction)
- **SAGA tries to address these issues**

OGF: top-down vs. bottom-up

- bottom-up often agrees on (semantic) LCD + backend specific extensions
- top-down usually focuses on semantics of application requirements

- bottom-up tends to be more powerful
- top-down tends to be simplier and more concise

- *we very much prefer top-down!*

SAGA

Simple API for Grid Applications

SAGA Design Principles

- **SAGA**: **S**imple **A**PI for **G**rid **A**pplications
- OGF approach to a uniform API layer (facade)
- **governing principle: 80:20 rule**
  simplicity versus control!
- **top-down approach:** use case driven!
  $\longrightarrow$ defines **application level** abstractions
- **extensible:** stable look & feel + API packages
- **influenced by:** DRMAA, GridRPC, OREP, JSDL, POSIX, GAT, CoG, LSF, Globus, ...
- API Specification is **Language Independent** (IDL)
  Renderings exist in C++, Python, Java
  Examples here are in C++

SAGA Intro: Example 1

SAGA: File Management

```
saga::filesystem::directory dir ("any://remote.host.net//data/");

if ( dir.exists ("a") && ! dir.is_dir ("a") )
{
  dir.copy ("a", "b", Overwrite);
}

list <saga::url> names = dir.find ("*-{123}.txt");

saga::filesystem::directory tmp  = dir.open_dir ("tmp/", Create);
saga::filesystem::file      file = dir.open     ("tmp/data.txt");
```

SAGA Intro: Example 1

- API is clearly POSIX (libc + shell) inspired
- where is my security??
- what is 'any://' ???

SAGA Intro: Example 2

SAGA: Job Submission

```
saga::job::description jd; // details left out
saga::job::service     js ("any://remote.host.net/");
saga::job::job         j = js.create_job (jd);

j.run ();

cout << "Job State: " << j.get_state () << endl;

j.wait ();

cout << "Retval " << j.get_attribute ("ExitCode") << endl;
```

SAGA Intro: Example 2’

SAGA: Job Submission

```
saga::job::service     js ("any://remote.host.net");
saga::job::job         j = js.run_job ("touch /tmp/touch.me");



cout << "Job State: " << j.get_state () << endl;

j.wait ();

cout << "Retval " << j.get_attribute ("ExitCode") << endl;
```

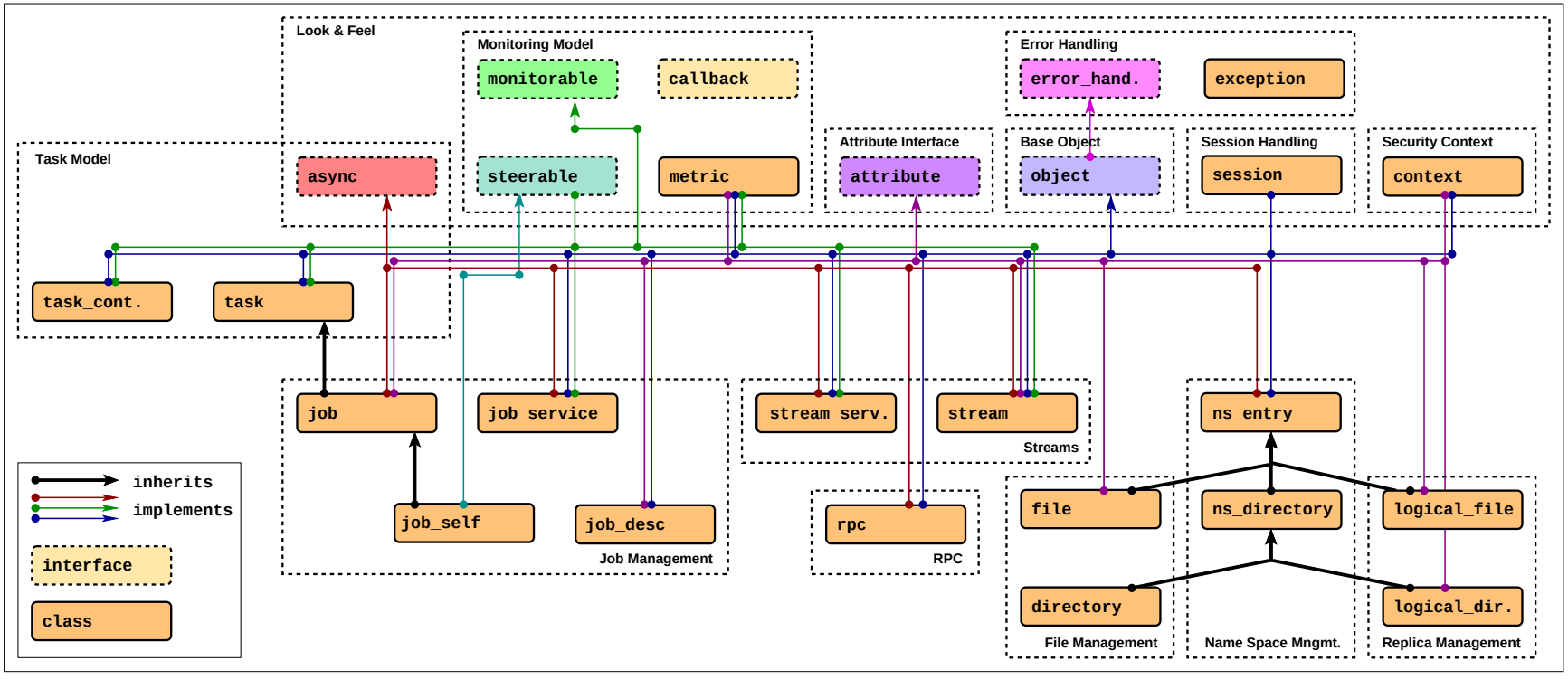
SAGA Intro: Example 2

- stateful objects!
- yet another job description langauge? `:-(`
- many hidden/default parameters (to keep call signatures small)
- 'any`://`' again!
- TIMTOWTDI (there is more than one way to do it)

SAGA Intro: 10.000 feet

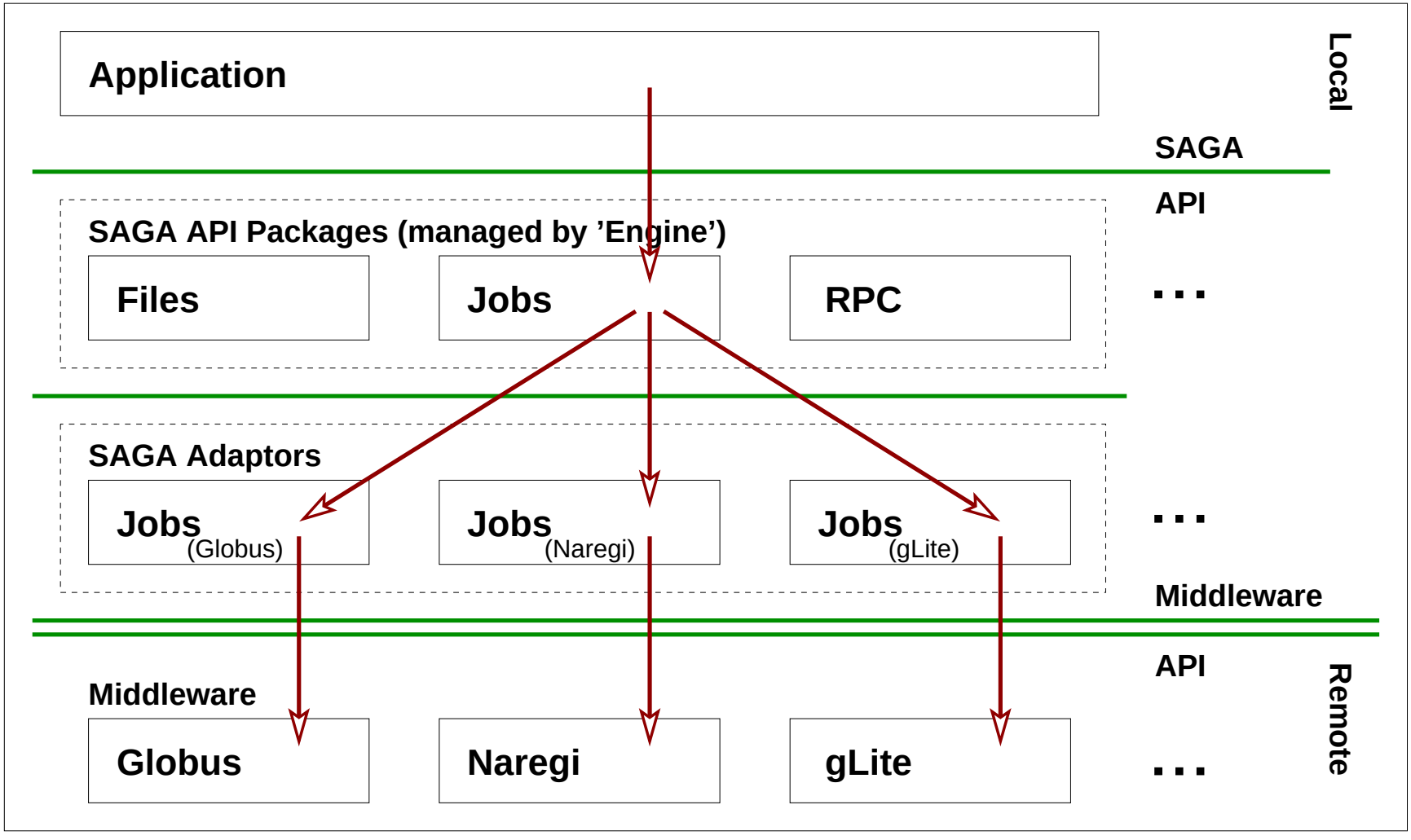
- **object oriented:** inheritance, interfaces
  very moderate use of templates though!
- functional and non-functional elements strictly separated
  - *non-functional API:* look & feel- orthogonal to functional API
    often not mappable to remote operations
  - *functional API:* API 'Packages' - extensible
    typically mappable to remote operations
- few inter-package dependencies - allows for partial implementations

SAGA: Class hierarchy

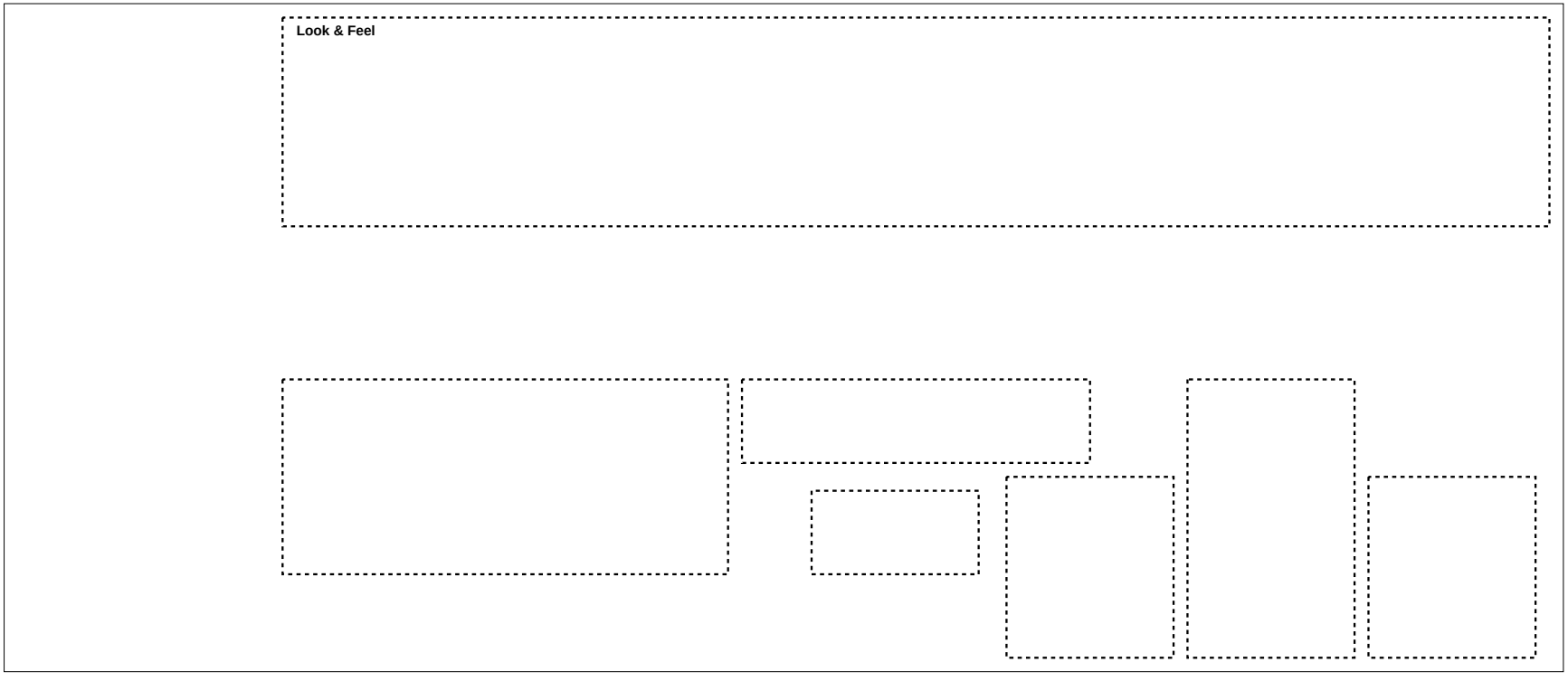


SAGA API Structure: look & feel (top) + API packages (bottom)

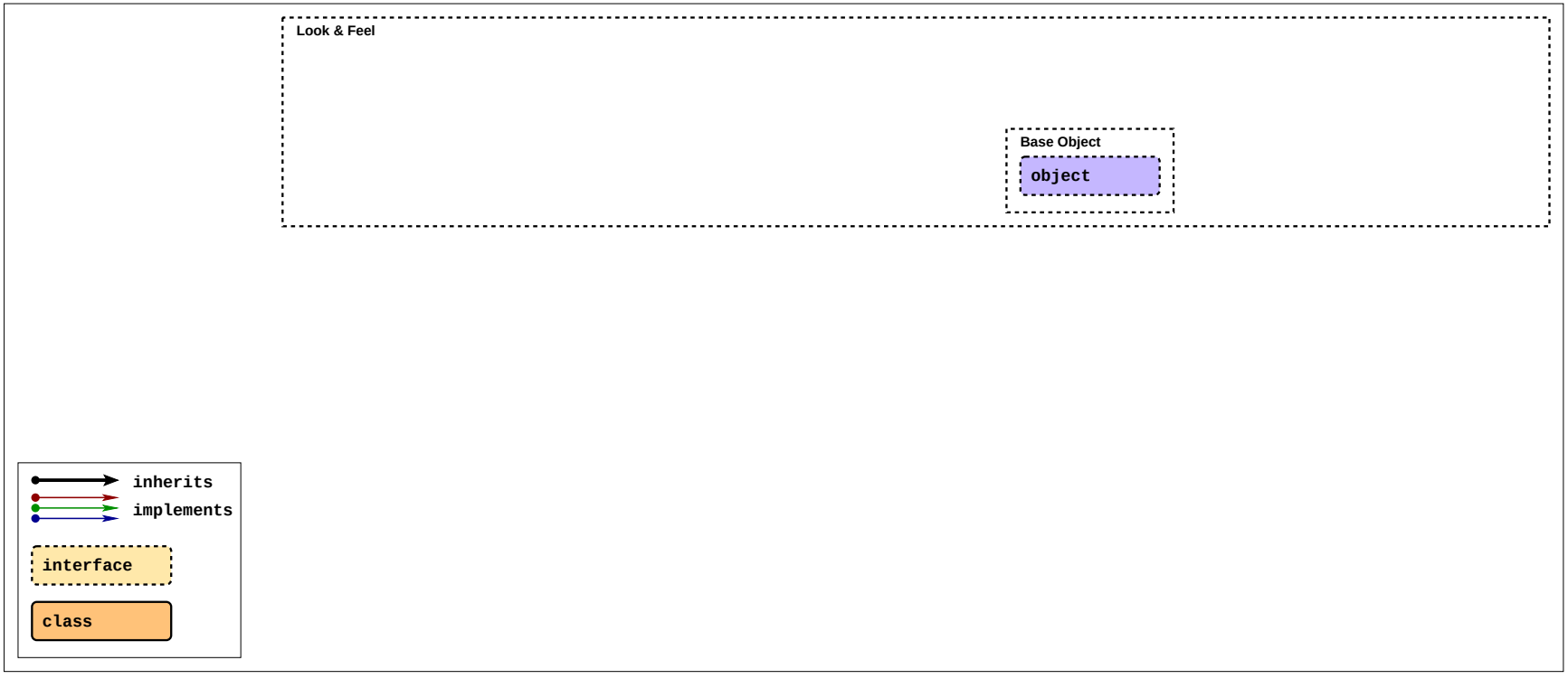
Implementation



SAGA: Class hierarchy



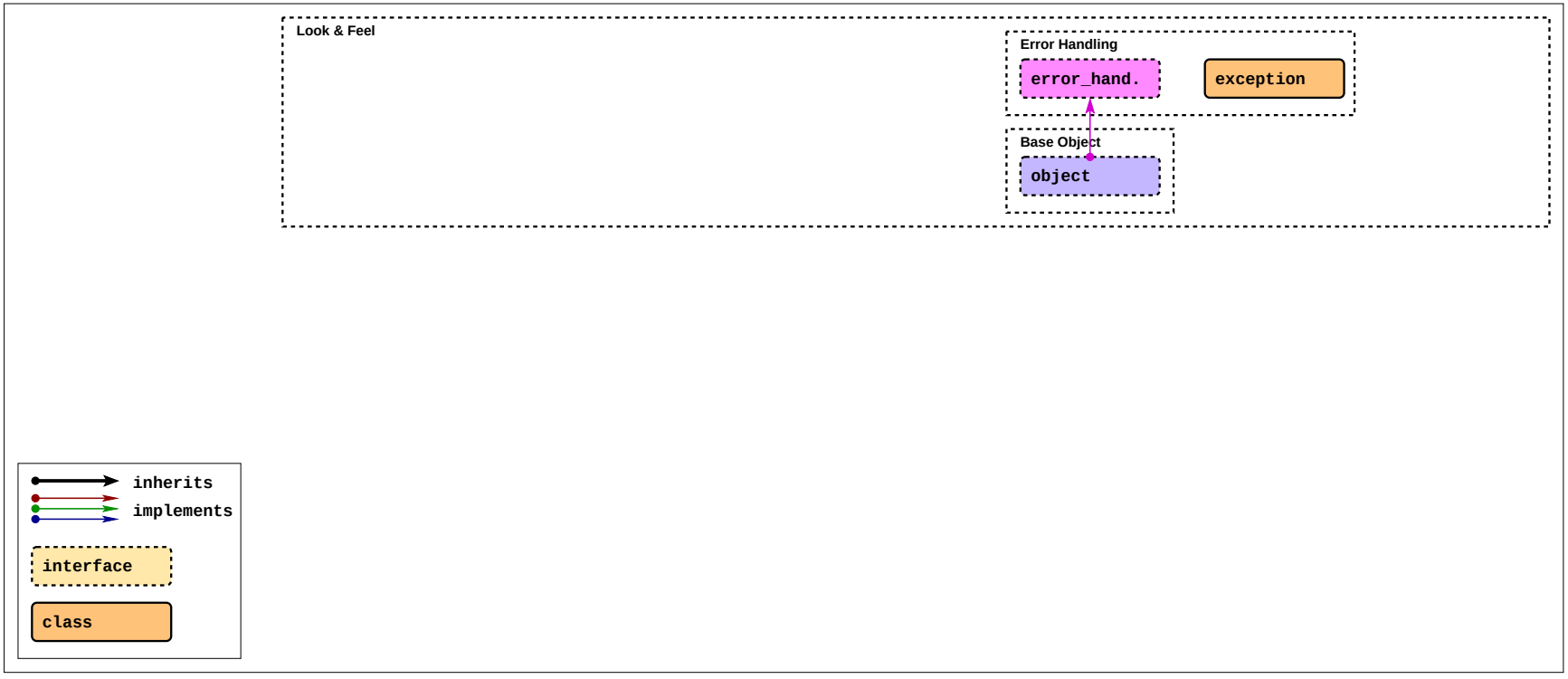
SAGA: Class hierarchy



SAGA Look & Feel:

`saga::object` allows for object uuids, `clone()` etc.

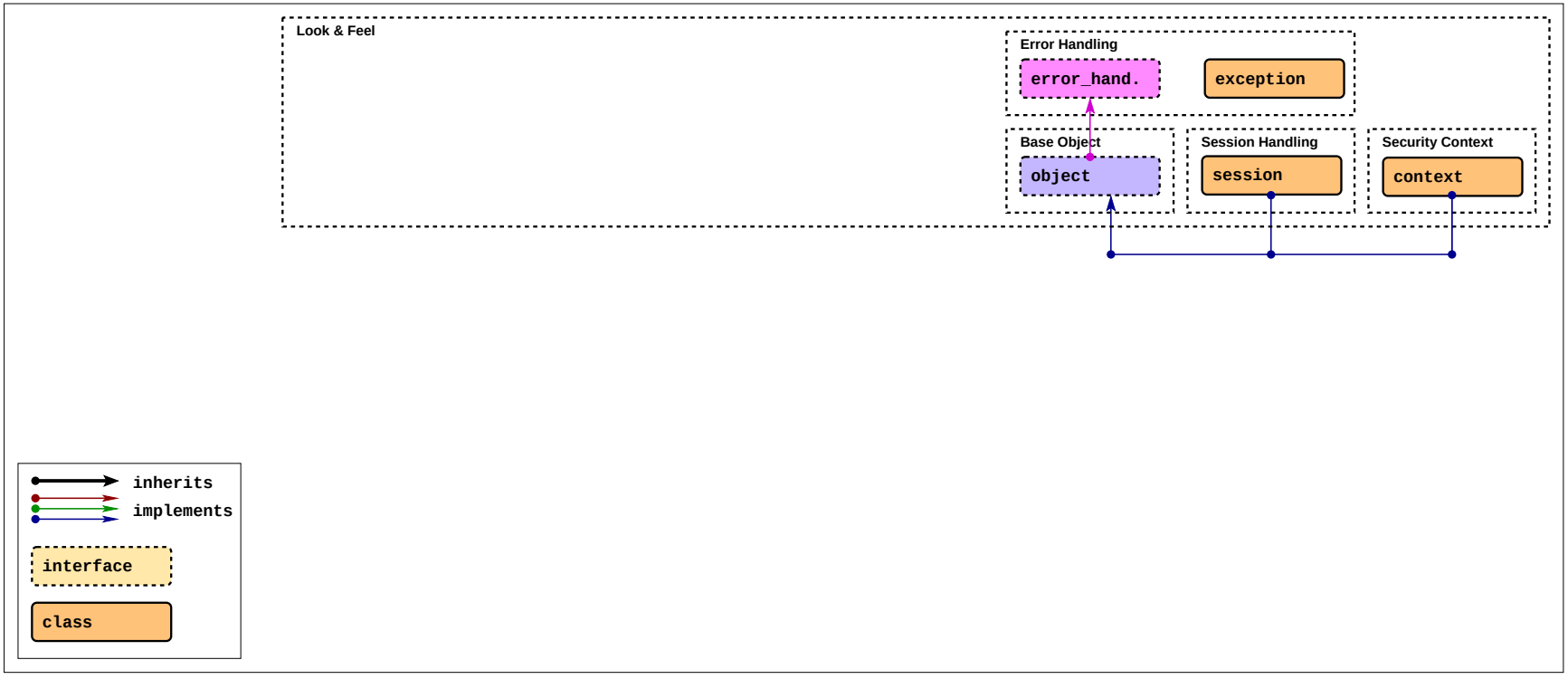
SAGA: Class hierarchy



SAGA Look & Feel:

errors are based on exceptions or error codes.

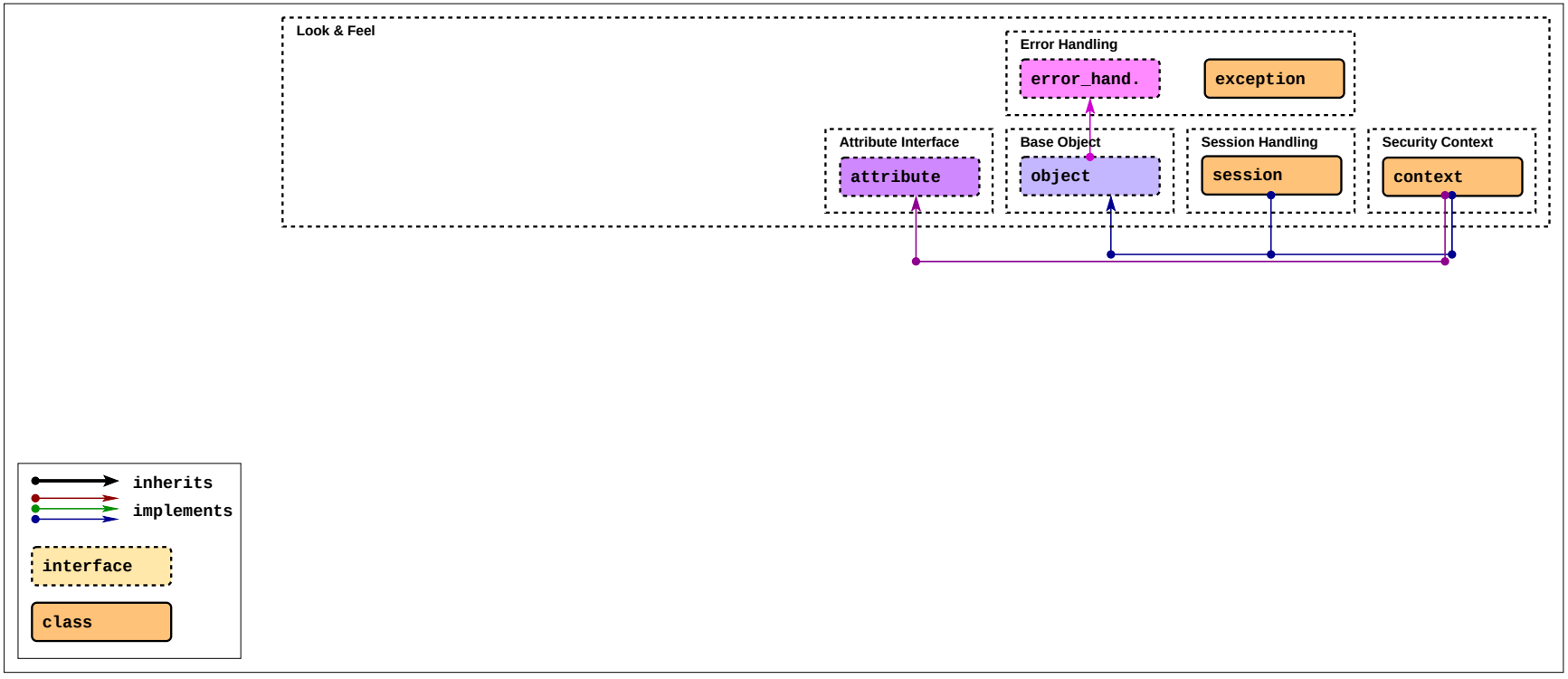
SAGA: Class hierarchy



SAGA Look & Feel:

session and credential management is hidden.

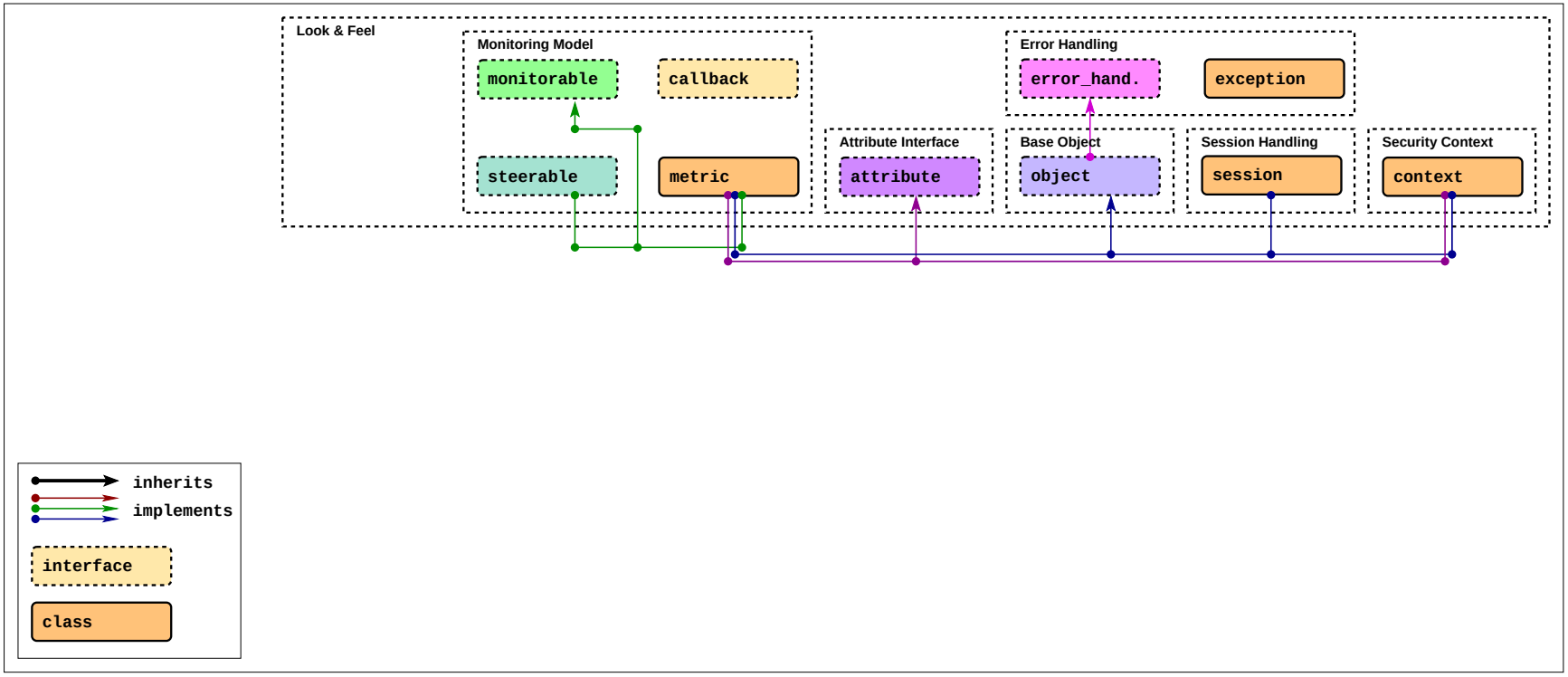
SAGA: Class hierarchy



SAGA Look & Feel:

Attribute interface for meta data.

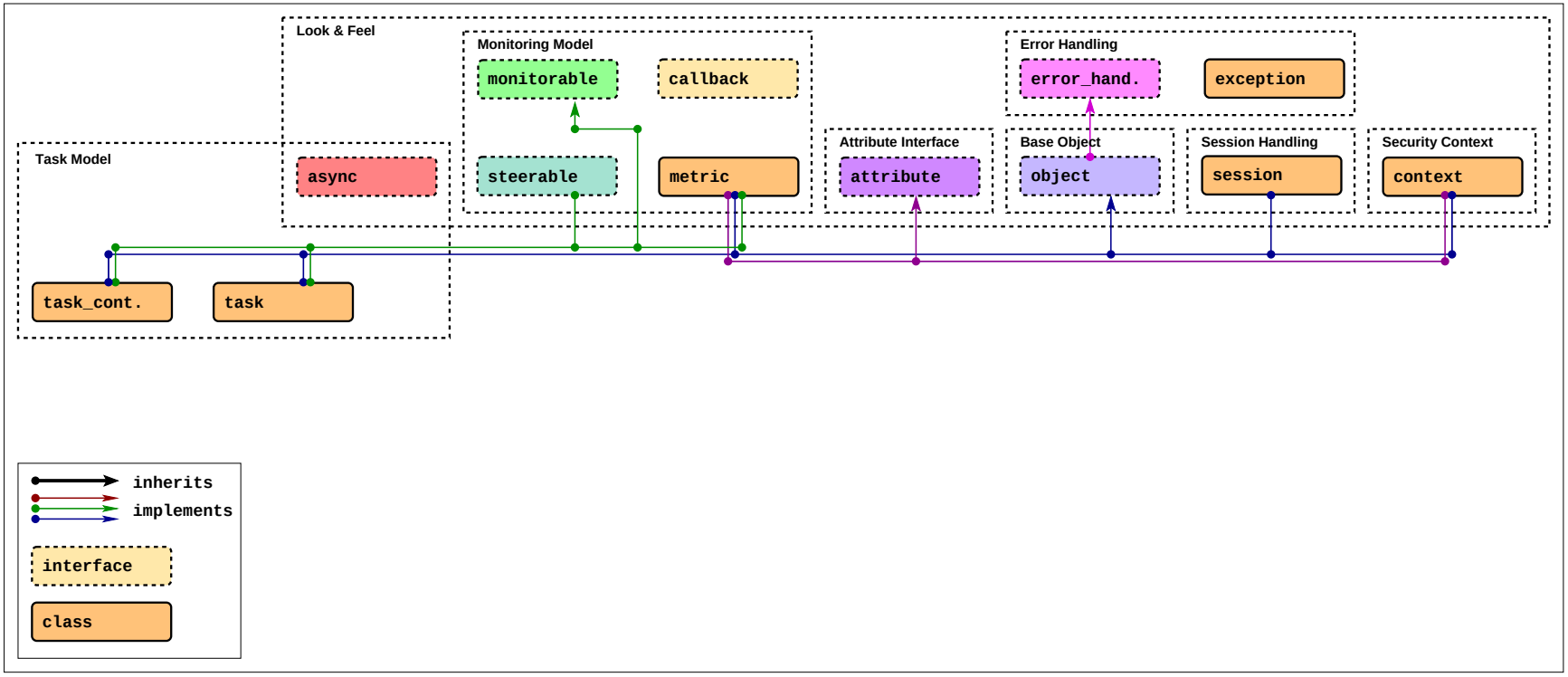
SAGA: Class hierarchy



SAGA Look & Feel:

Monitoring includes asynchronous notifications.

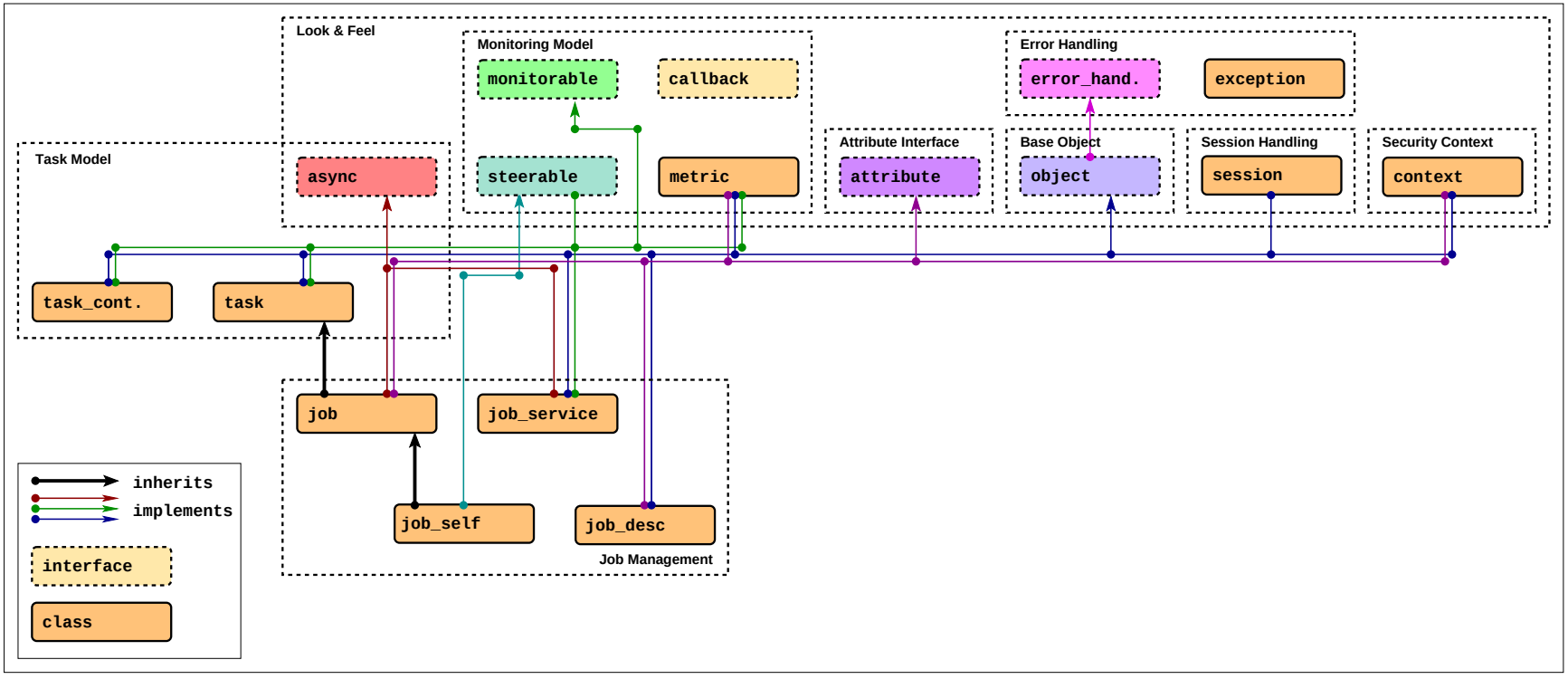
SAGA: Class hierarchy



SAGA Look & Feel:

the task model adds asynchronous operations.

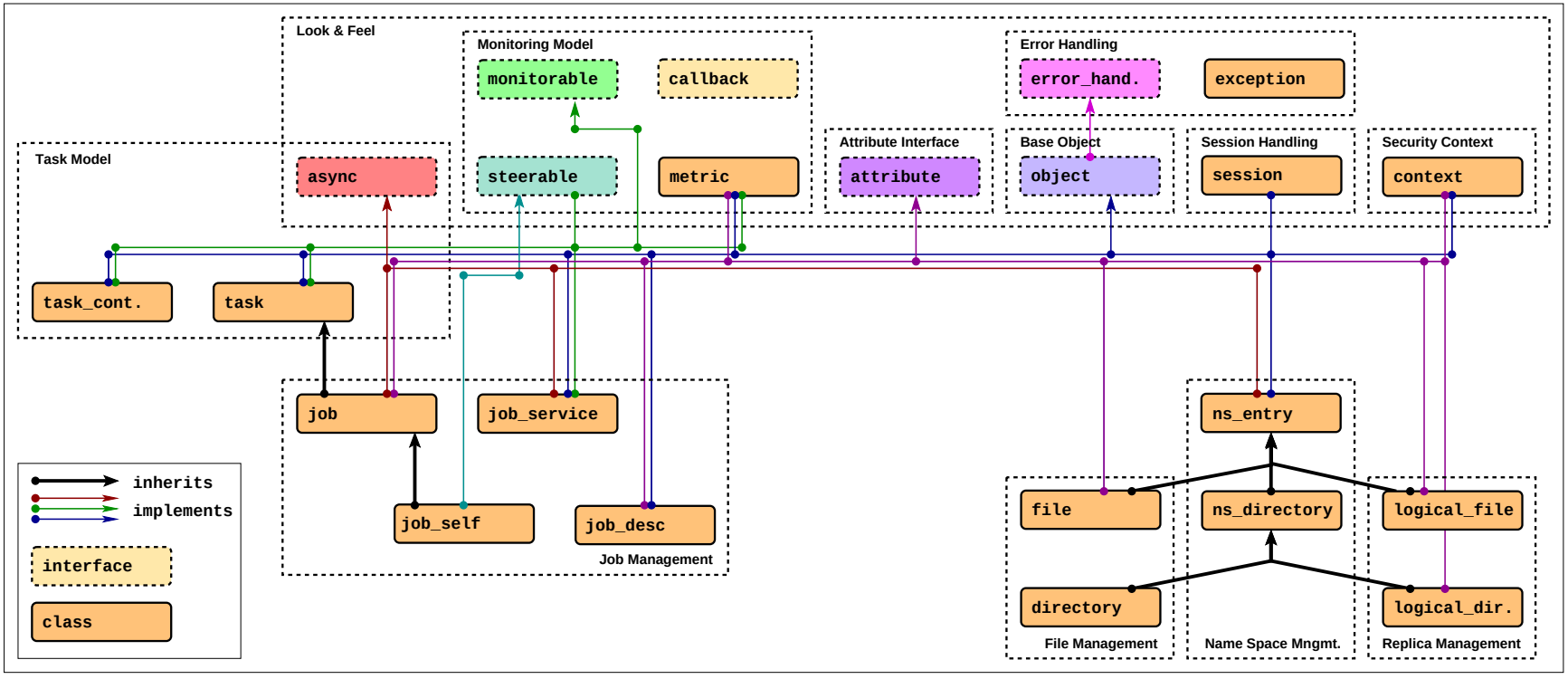
SAGA: Class hierarchy



SAGA API Package ’job’:

create and manage remote processes.

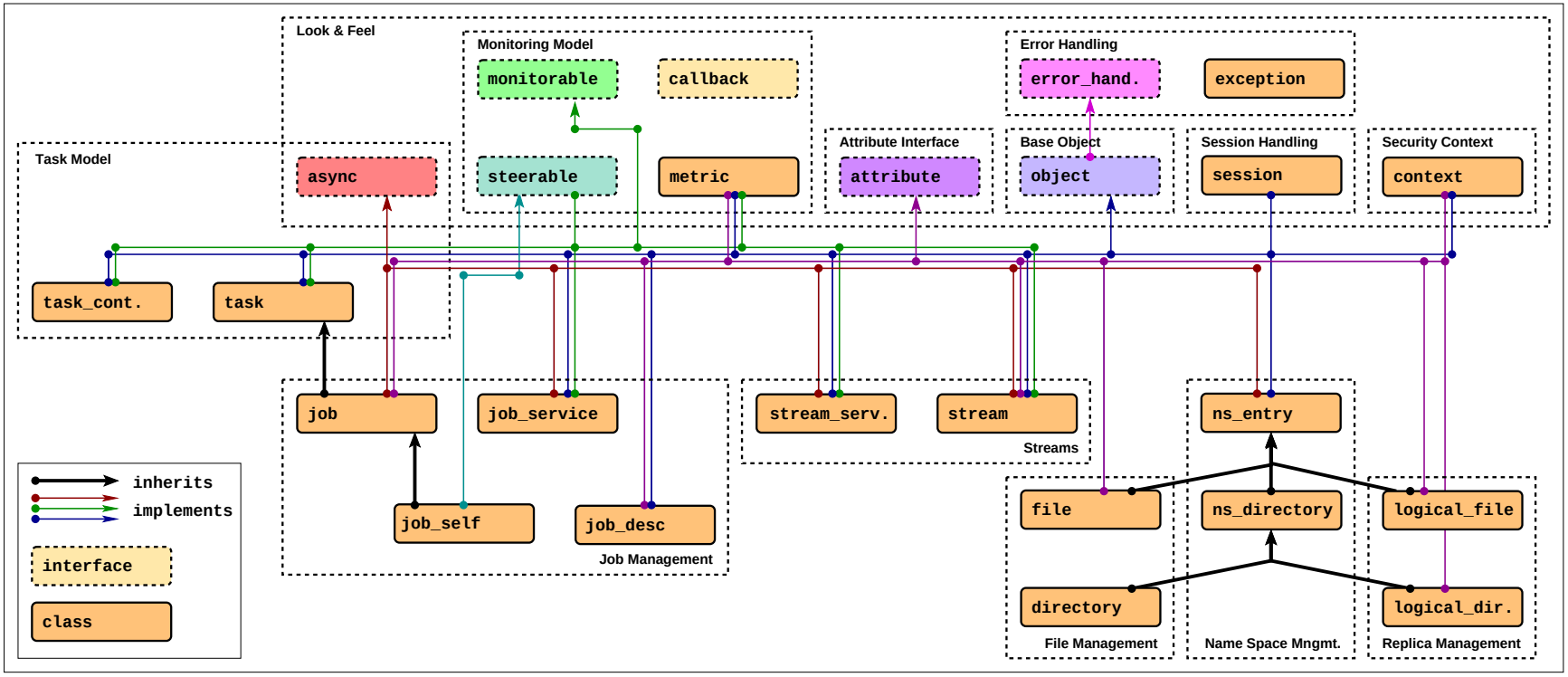
SAGA: Class hierarchy



SAGA API Package 'name_spaces':

manage files, replicas, etc.

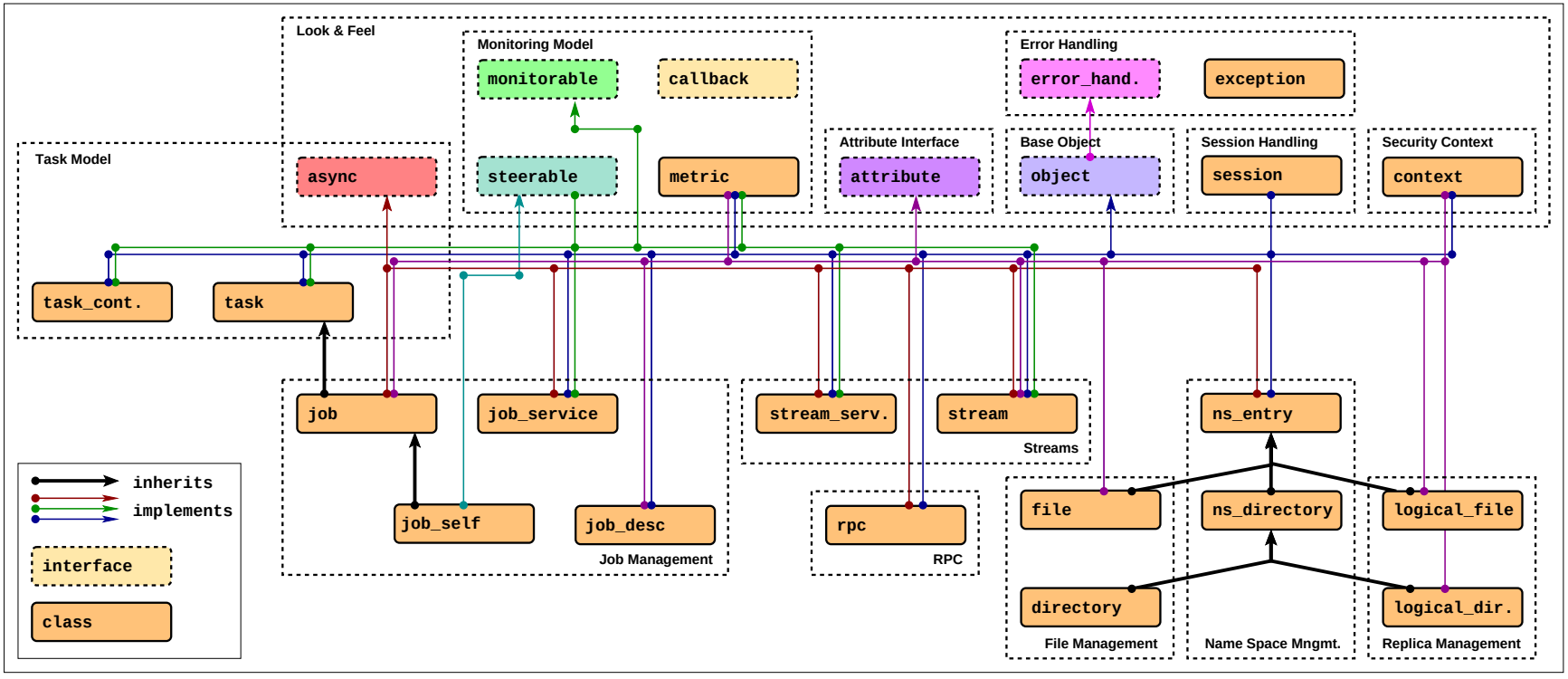
SAGA: Class hierarchy



SAGA API Package 'stream':

SAGA rendering of BSD streams.

SAGA: Class hierarchy

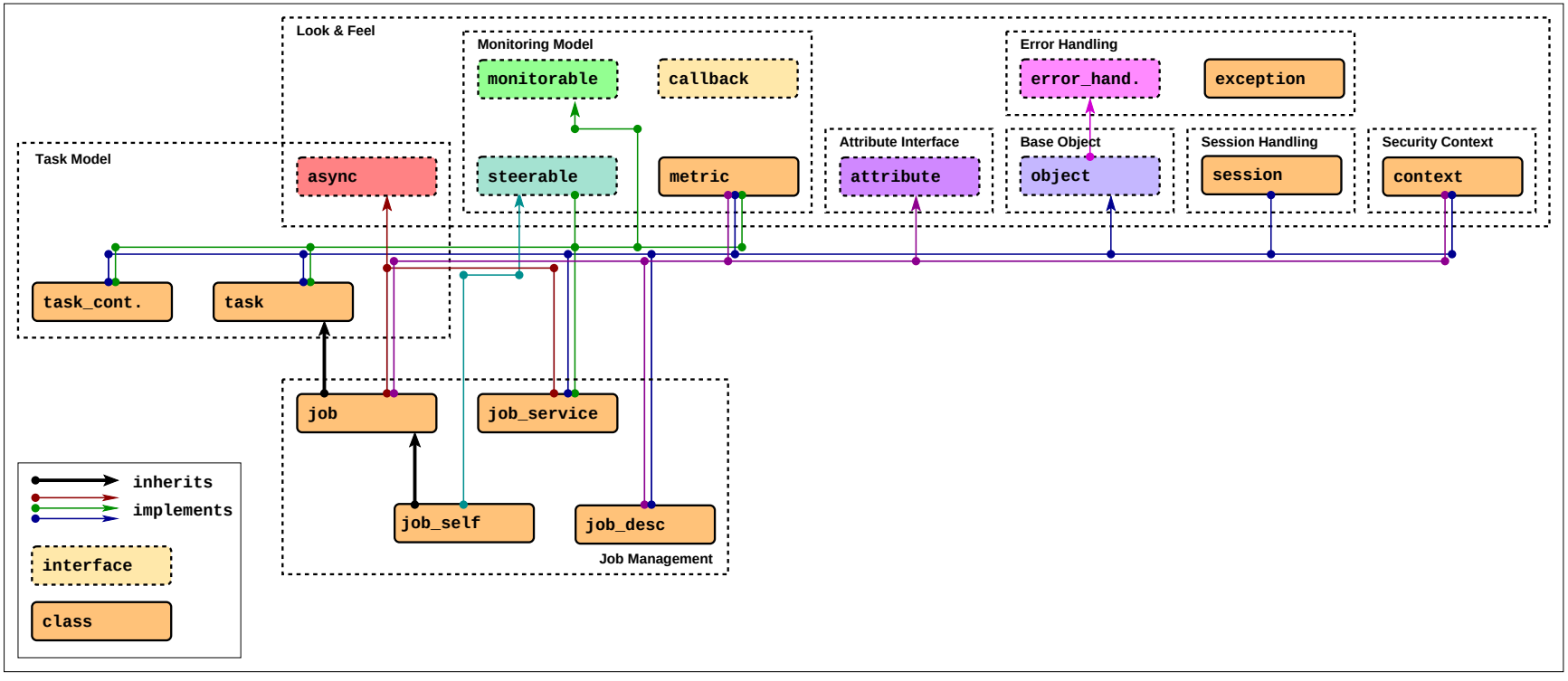


SAGA API Package 'rpc':

remote procedure calls.

Functional API Packages

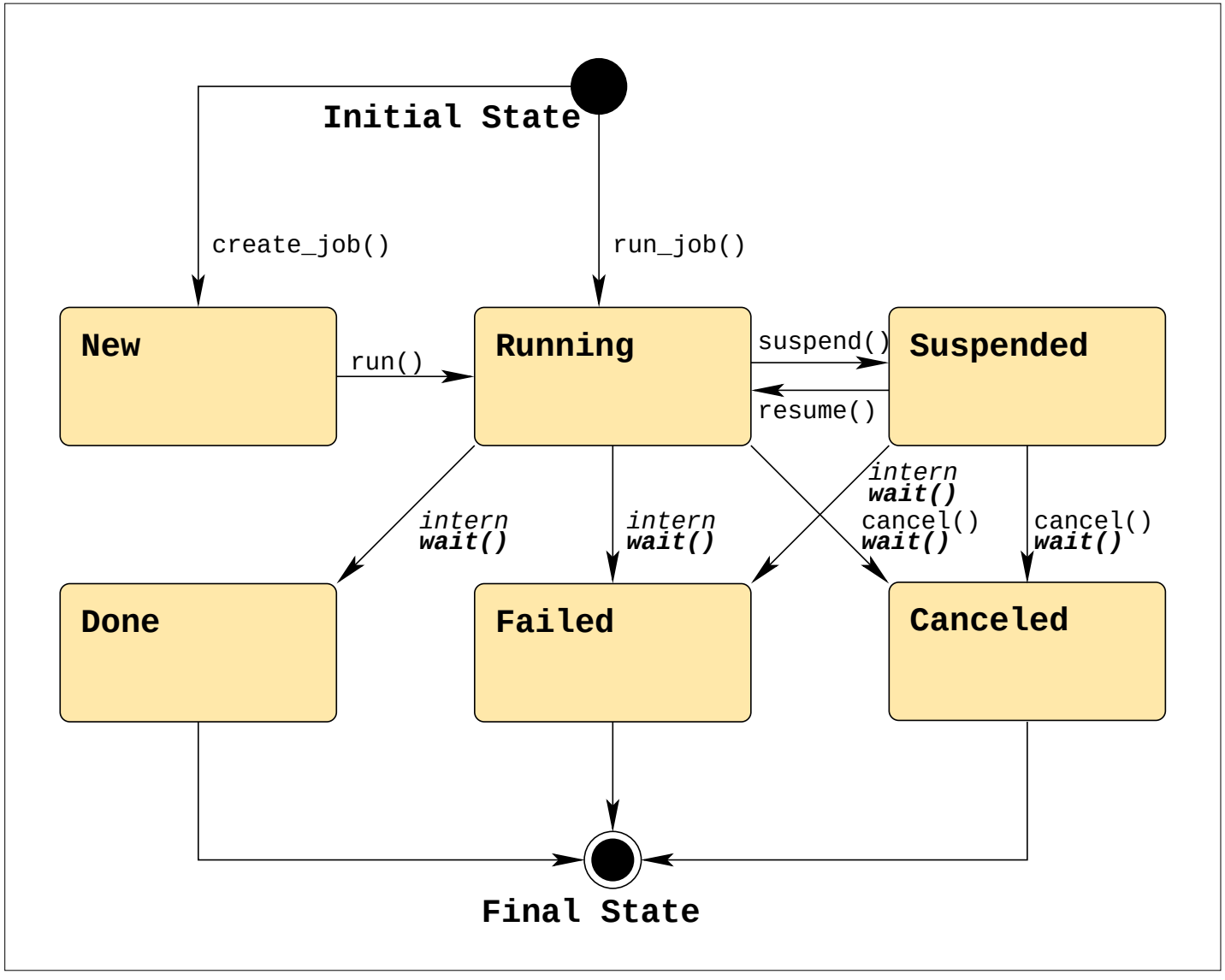
SAGA: Jobs


Look & Feel
Monitoring Model
monitorable
callback
Error Handling
error_hand.
exception
Task Model
async
steerable
metric
Attribute Interface
attribute
Base Object
object
Session Handling
session
Security Context
context
task_cont.
task
job
job_service
job_self
job_desc
Job Management
inherits
implements
interface
class

SAGA: Jobs - Overview

- `job_service` uses `job_description` to create `job` instances
- `job_description` attributes are based on JSDL
- state model is based on / synced with BES
- `job_self` represents the SAGA application
- job submission and management, but no resource discovery, job dependencies, or workflows

SAGA: Job States



SAGA Examples: Jobs

job submission

```
saga::job::service     js ("gram://headnode.gram.net");
saga::job::job         j = js.run_job ("/bin/sleep 10",
                                       "clusternode-2.gram.net");


cout << "Job State: " << j.get_state () << endl;

j.wait ();

cout << "Retval " << j.get_attribute ("ExitCode") << endl;
```

SAGA Examples: Jobs

job submission

```
saga::job::description jd;
saga::job::service     js ("gram://remote.host.net");
saga::job              j = js.create_job (jd);

j.run ();

cout << "Job State: " << j.get_state () << endl;

j.wait ();

cout << "Retval " << j.get_attribute ("ExitCode") << endl;
```

SAGA Examples: Jobs

jobs (cont.)

```
j.run     ();
j.wait    ();
j.cancel  ();

j.suspend ();
j.resume  ();

j.signal     (SIGUSR1);
j.checkpoint ();
j.migrate    (jd);
```

SAGA Examples: Job Descr.

job description - JSDL based

```
saga::job::description jd;

jd.set_attribute ("Executable",       "/bin/tail");
jd.set_attribute ("WorkingDirectory", "data/");
jd.set_attribute ("Cleanup",          "False");

// pseudo code *blush*
jd.set_vector_attribute ("Arguments",    ["-f", "my_log"]);
jd.set_vector_attribute ("Environment",  ["TMPDIR=/tmp/"]);
jd.set_vector_attribute ("FileTransfer", ["my_log >> all_logs"]);
```

SAGA Job Description

SAGA JD attributes:

Executable	Arguments	Environment
CandidateHosts	SPMDVariation	TotalCPUCount
NumberOfProcesses	ProcessesPerHost	ThreadsPerProcess
WorkingDirectory	*Interactive*	Cleanup
Input	Output	Error
JobStartTime	WallTimeLimit	TotalCPUTime
TotalPhysicalMemory	CPUArchitecture	OperatingSystemType
Queue	JobProject	*JobContact*
FileTransfer		

SAGA Job Description

- leaning heavily on **JSDL**, but flat
- borrowing from DRMAA
- mixes hardware, software and scheduling attributes!
- cannot be extended
- no support for 'native' job descriptions (RSL, JDL, ...)
- only '`Executable`' is required
- backend MAY ignore unsupported keys!

```
cd /tmp/data && rm -rf *
```

SAGA Example: job service

job service

```
saga::job::service js ("gram://remote.host.net/");

vector<string> ids = js.list (); // list known jobs

while ( ids.size () )
{
  string id = ids.pop_back ();

  saga::job j = js.get_job (id);  // reconnect to job

  cout << id << " : " << j.get_state () << endl;
}
```

SAGA Job Service

- represents a specific job submission endpoint
- job states are maintained on that endpoint (usually)
- full reconnect may not be possible (I/O streaming)
- lifetime of state up to backend
- reconnected jobs may have different job description (lossy translation)

SAGA Examples: Job 'Self'

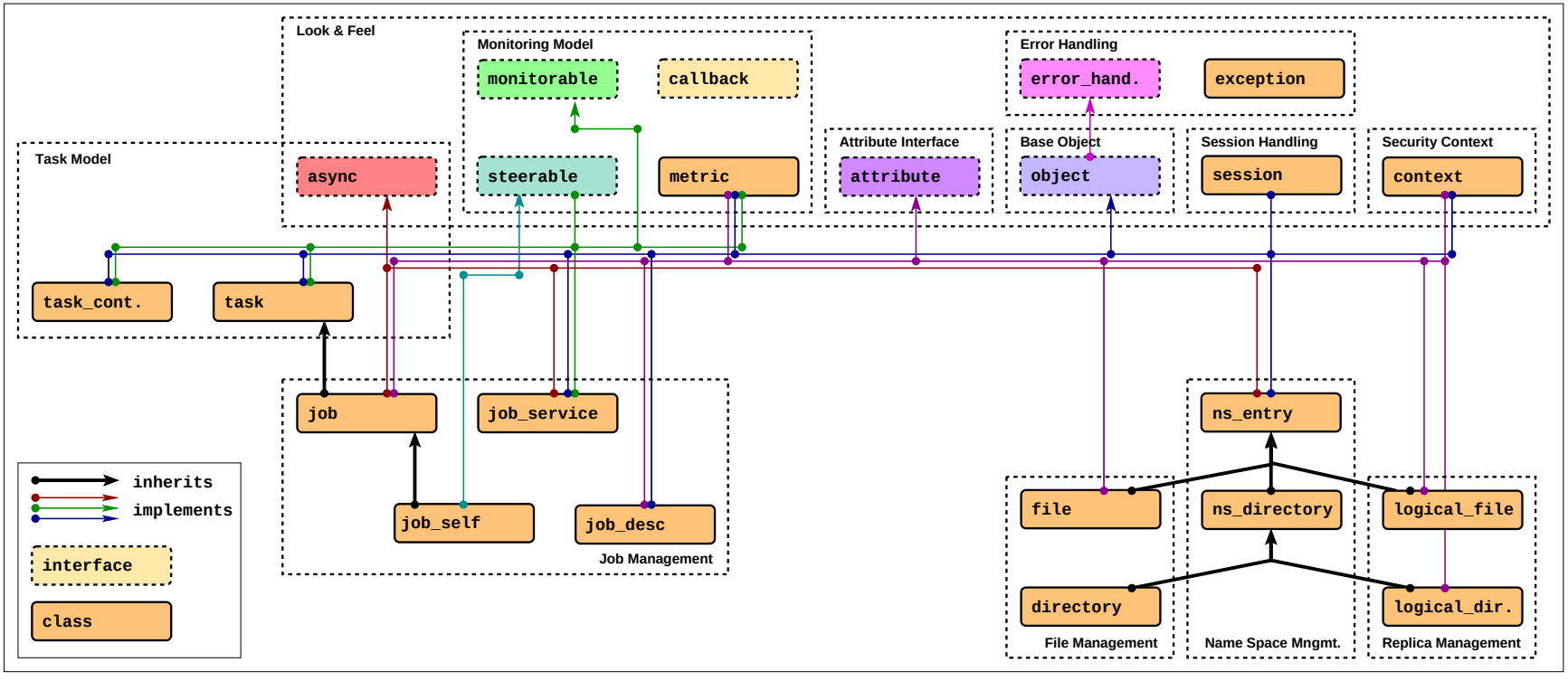
jobs (cont.)

```
saga::job::self self = js.get_self ();

self.signal     (SIGUSR1);
self.checkpoint ();
self.migrate    (jd);

self.wait   ();   // blocks forever :-P
self.cancel ();
```

SAGA Job Self

- represents the **calling application instance**
- `self.signal (SIGUSR1);`
  is different from
  `::kill (::getpid (), SIGUSR1);`
  as it goes over the job manager (accounting, logging, cleanup...)
- '`job::self`' is the only SAGA object to which metrics can be added (details later...)

SAGA: Name Spaces etc.



SAGA: Name Spaces

- interfaces for managing entities in name spaces
- files, replicas, information, resources, steering parameter, checkpoints, . . .
- manages hierarchy (mkdir, cd, ls, . . .)
- manages NS entries as opaque (copy, move, delete, ...)

SAGA: Files

- implements name space interface, and adds access to content of NS entries (files)
- Posix oriented: read, write seek
- Grid optimizations: scattered I/O, pattern based I/O, extended I/O

SAGA: Replicas

- implements name space interface, and adds access to properties of NS entries (logical files / replicas)
- O/REP oriented: list, add, remove replicas; manage meta data
- Grid optimizations are hidden (replica placement strategies, consistency and version management, . . .)

SAGA: Adverts

- implements name space interface, and adds access to arbitrary key/value pairs on each entry.
- entries also allow to store a **serialized** SAGA Object!
- utterly useful for application bootstrapping, communication between different application modules, application persistency, etc etc.
- **not part of the SAGA Specification**, but is getting standardized as an extension

SAGA Examples: NameSpaces

name space management

```
saga::name_space::directory d ("ssh://remote.host.net//data/");

if ( d.is_entry ("a") && ! d.is_dir ("a") )
{
  d.copy ("a", "../b");
  d.link ("../b", "a", Overwrite);
}

list <saga::url> names = d.find ("*-{123}.text.");

saga::name_space::directory tmp  = d.open_dir ("tmp/data/1",
                                   saga::name_space::CreateParents);
saga::name_space::entry     data = tmp.open   ("data.txt");

data.copy ("data.bak", Overwrite);     // uses cwd
```

SAGA Name Spaces

- name space entries are opaque: the name space package can never look inside
- directories are entries (inheritance)
- **inspection:**
  `get_cwd, get_url, get_name, exists, is_entry, is_dir, is_link, read_link`
- **manipulation:**
  `create (c'tor, open), copy, link, move, remove`
- **permissions:**
  `permissions_allow, permissions_deny`
- wildcards are supported (remember POSIX influence...)

SAGA Examples: Files

file access

```
saga::filesystem::file f ("any://remote.host.net/data/data.bin");

char mem[1024];
saga::mutable_buffer buf (mem);

if ( f.get_size () >= 1024 )
{
  buf.set_data (mem + 0, 512);
  f.seek (512, saga::filesystem::Start);
  f.read (buf);
}

if ( f.get_size () >= 512 )
{
  buf.set_data (mem + 512, 512);
  f.seek (0, saga::filesystem::Start);
  f.read (buf);
}
```

SAGA Filesystem

- provides access to the **content** of filesystem entries (sequence of bytes)
- saga buffers are used to wrap raw memory buffers
- saga buffers can be allocated by the engine
- several incarnations of read/write: posix style, scattered, pattern based

SAGA Name Spaces: Flags

flags

```
enum flags {
  None           = 0,
  Overwrite      = 1,
  Recursive      = 2,
  Dereference    = 4,
  Create         = 8,
  Exclusive      = 16,
  Lock           = 32,
  CreateParents  = 64,
  Truncate       = 128,   // not on name_space
  Append         = 256    // not on name_space
  Read           = 512,
  Write          = 1024,
  ReadWrite      = 1536   // Read | Write
  Binary         = 204    // only on filesystem
}
```

SAGA Examples: Replicas

replica management

```
saga::replica::directory dir ("raptor://remote.host.net/data/");

if ( dir.is_entry ("a") || dir.is_link ("a") )
{
  dir.copy ("a", "../b");
  dir.link ("../b", "a");
}

saga::replica::file file = dir.open ("tmp/data.txt");
list <string>  locations = file.list_locations ();

file.replicate ("gridftp://other.host.net/tmp/a.dat");
```

SAGA Replica

- provides access to the **content** of replica system entries (list of physical locations, plus attributes)
- saga attribute interface is used for entry meta data. Meta data are maintained by application, and/or backend.
- `replicate()` creates a new copy, and adds new location to list

SAGA Examples: Replicas

replica meta data

```
saga::replica::directory dir ("raptor://remote.host.net/data/");

list <saga::url> files = dir.find ("*", "type=jpg");

while ( file.size () )
{
  saga::logical_file lf (file.pop_front ());

  lf.replicate ("file://localhost/data/images/",
                saga::replica::Overwrite);
}
```

SAGA Adverts

- persistent storage of application level information
- semantics of information defined by application
- allows storage of serialized SAGA objects (object persistency)

SAGA Examples: Adverts

Adverts

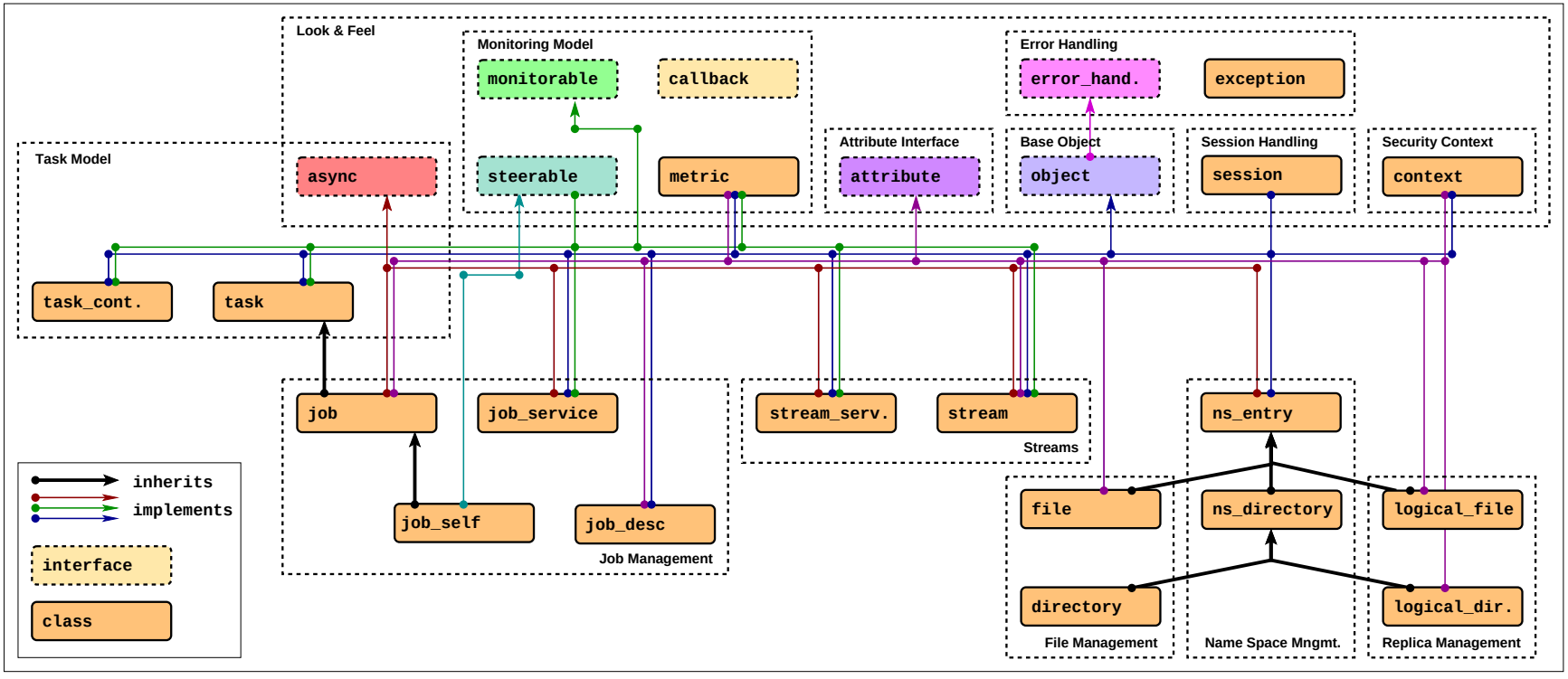
```
saga::advert::directory todo ("any//remote.host.net/my_tasks/");

// pseudo vector code
list <saga::url> urls = todo.find ("*", ["priority=urgent"]);

while ( urls.size () )
{
  saga::advert ad (urls.pop_front ());

  std::cout << ad.get_attribute ("description") << std::endl;
}
```

SAGA Examples: Adverts

Persistent SAGA Objects

```
saga::file   f  (url);
saga::advert ad ("any//remote.host.net/files/my_file_ad", Create);

ad.store_object (f);

-----------------------------------------------------------------

saga::advert ad ("any//remote.host.net/files/my_file_ad");
saga::file   f = ad.retrieve_object ();
```

SAGA: Streams



SAGA Examples: Streams

stream server

```
saga::stream_service ss ("tcp://localhost:1234");

saga::stream_client sc = ss.serve ();

sc.write ("Hello client", 13);
```

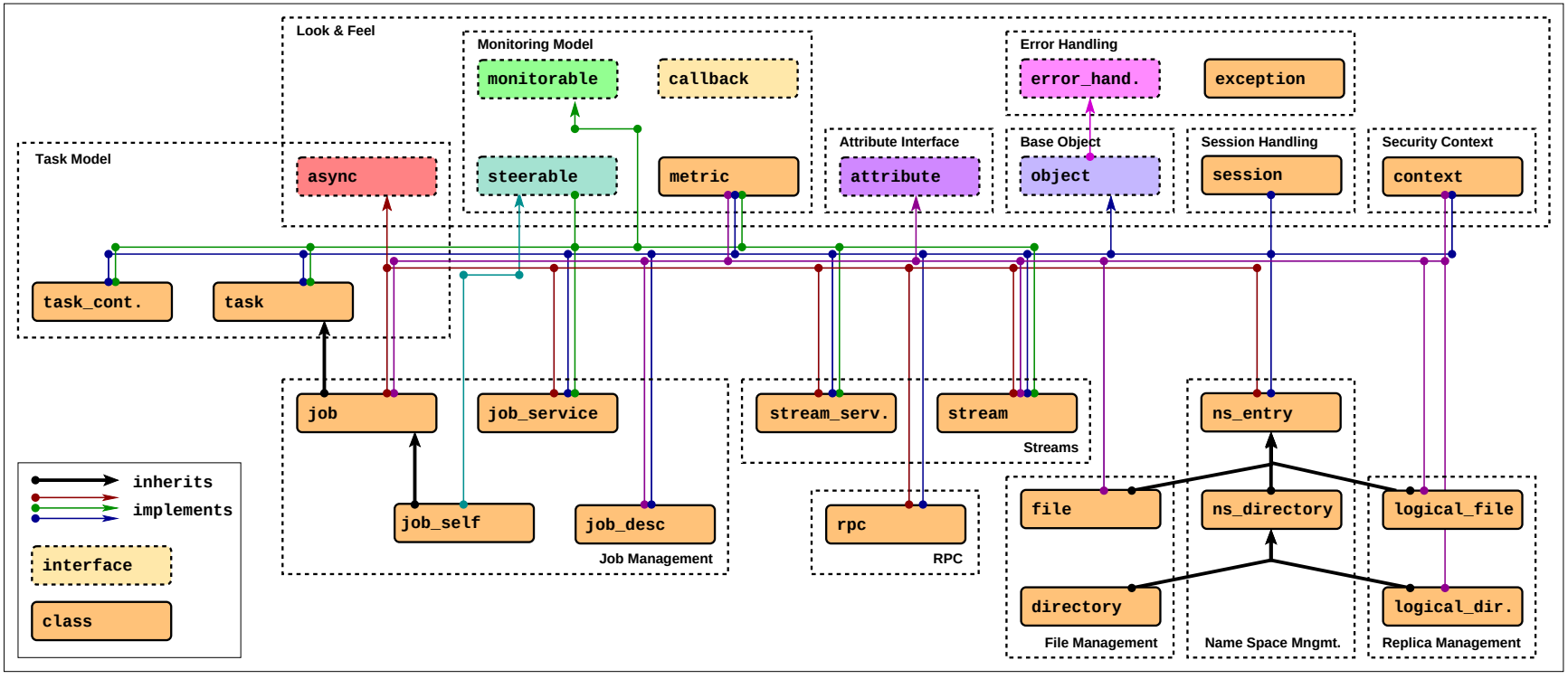
stream client

```
char buf [13];
saga::stream_client sc ("tcp://remote.host.net:1234");

sc.connect ();
sc.read    (buf, 13);

cout << buf << endl;
```

SAGA: Streams

- simple and BSD socket oriented
- not supposed to replace MPI etc, but allows for simple application level communication
- will be superceded by message package

SAGA: RPC


Look & Feel
Monitoring Model
monitorable
callback
Error Handling
error_hand.
exception
Task Model
async
steerable
metric
Attribute Interface
attribute
Base Object
object
Session Handling
session
Security Context
context
task_cont.
task
job
job_service
stream_serv.
stream
Streams
ns_entry
inherits
implements
interface
class
job_self
job_desc
Job Management
rpc
RPC
file
ns_directory
logical_file
directory
logical_dir.
File Management
Name Space Mngmt.
Replica Management

SAGA Examples: RPC

remote procedure call

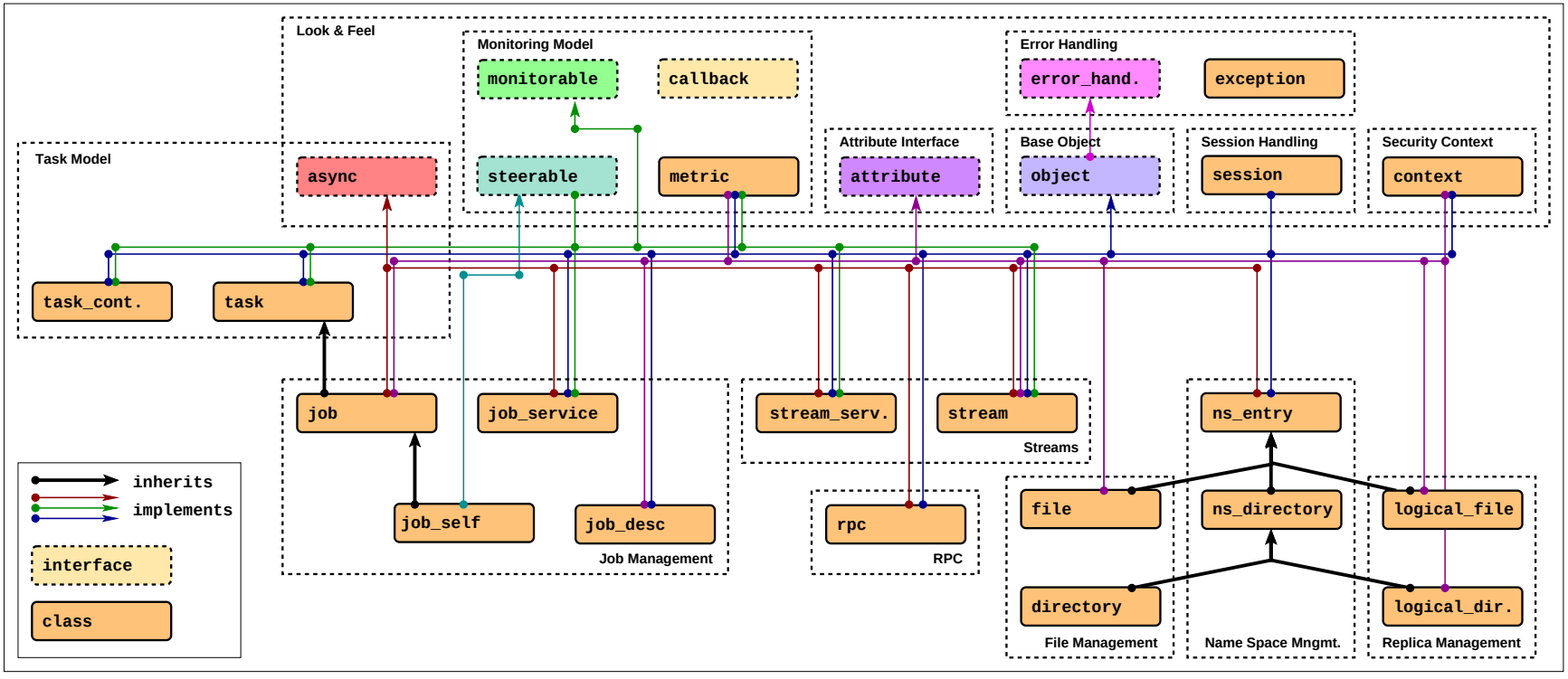
```
saga::rpc rpc ("ninfg://remote.host.net:1234/random");

list <saga::rpc::parameter> params;
params.push_back (new saga::rpc::parameter (Out, 10));

rpc.call (params);

cout << "found random number: " << ::atoi (param.buffer) << endl;

delete (params.pop_front ());
```

SAGA: RPC

- maps GridRPC standard into the SAGA look & feel
- parameters are stack of structures (similar to scattered I/O)
- future revision will work on optimized data handling

Non-Functional API Packages

SAGA: Session and Context



SAGA Examples: Session

default sessions

```
saga::ns_dir dir ("any://remote.host.net//data/");

if ( dir.is_entry ("a") && ! dir.is_dir ("a") )
{
  dir.copy ("a", "../b");
  dir.link ("../b", "a", Overwrite);
}

list <saga::url> names = dir.find ("*-{123}.text.");

saga::name_space::directory  tmp = dir.open_dir ("tmp/");
saga::name_space::entry    entry = tmp.open     ("data.txt");

entry.copy ("data.bak", Overwrite);
```

SAGA Examples: Session

context management

```
saga::context c1 (saga::context::X509);
saga::context c2 (saga::context::X509);

c2.set_attribute ("UserProxy", "/tmp/x509up_u123.special");

saga::session s;

s.add_context (c1);
s.add_context (c2);

saga::name_space::dir dir (s, "any://remote.host.net/data/");
```

SAGA: Session Management

- by default hidden (default session is used)
- session is identified by lifetime of security credentials and by objects in this session (jobs etc.)
- session is used on object creation (optional)
- `saga::context` is used to attach security tokens to a session
- the default session has default contexts

SAGA Examples: Session

session inheritance

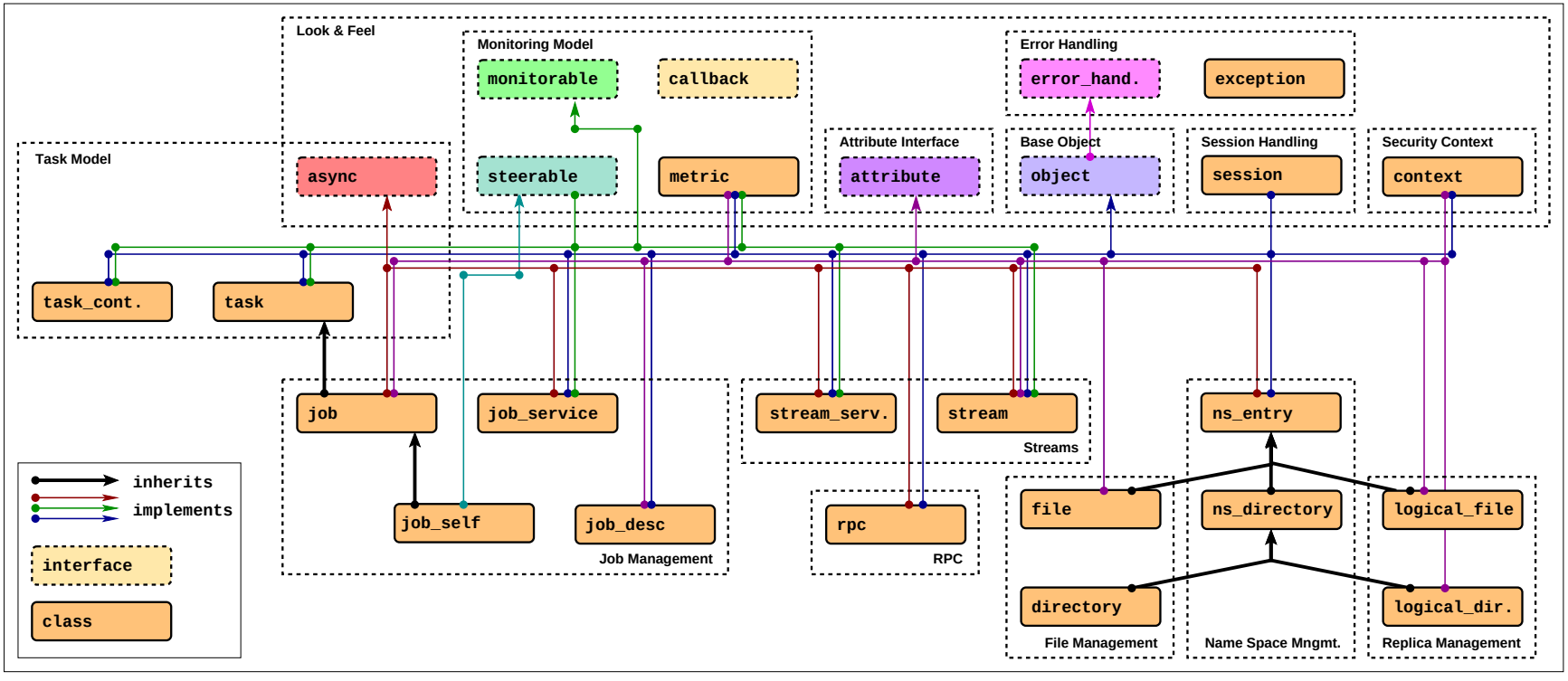
```
saga::dir  dir (s, "gridftp://remote.host.net/data/");

saga::file file = dir.open ("data.bin");

s.remove_context (c1);
s.remove_context (c2);

file.copy ("data.bin.bak");   // works - state is sticky!
```

SAGA Examples: Contexts

authorization

```
// server side code
saga::stream_service ss ("tcp://localhost:1234");

saga::stream_client sc = ss.serve ();

saga::context c = sc.get_context ();

if ( c.get_type == Globus &&
     c.attribute_equals ("RemoteID", "O=MyCA, O=MyOrg, CN=Joe") )
{
  sc.write ("welcome!", 9);
}
else
{
  sc.write ("bugger off!", 12);
  sc.close ();
}
```

SAGA: Monitoring



SAGA: Monitoring

- monitoring of Grid entities (jobs, files, . . .)
- monitoring of interactions (task state, notification, . . .)
- `monitorables` have metrics
- `metrics` can be pulled, or subscribed to (callbacks)
- some metrics can be written (basic steering)

SAGA Examples: Monitoring

pull monitoring

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

j.run ();

saga::metric m = j.get_metric ("MemoryUsage");

while ( 1 )
{
  cout << "Memory Usage: " << m.get_value () << endl;
  sleep (1);
}
```

SAGA Examples: Monitoring

callbacks

```
class my_cb : public saga::callback
{
  public:
    bool cb (saga::monitorable obj,
             saga::metric      m,
             saga::context     c)
    {
      cout << "Memory Usage: " << m.get_value () << endl;
      return (true);
    }
};

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

saga::metric m = j.get_metric ("MemoryUsage");
m.add_callback ("MemoryUsage", cb);
```

SAGA Examples: Monitoring

callbacks

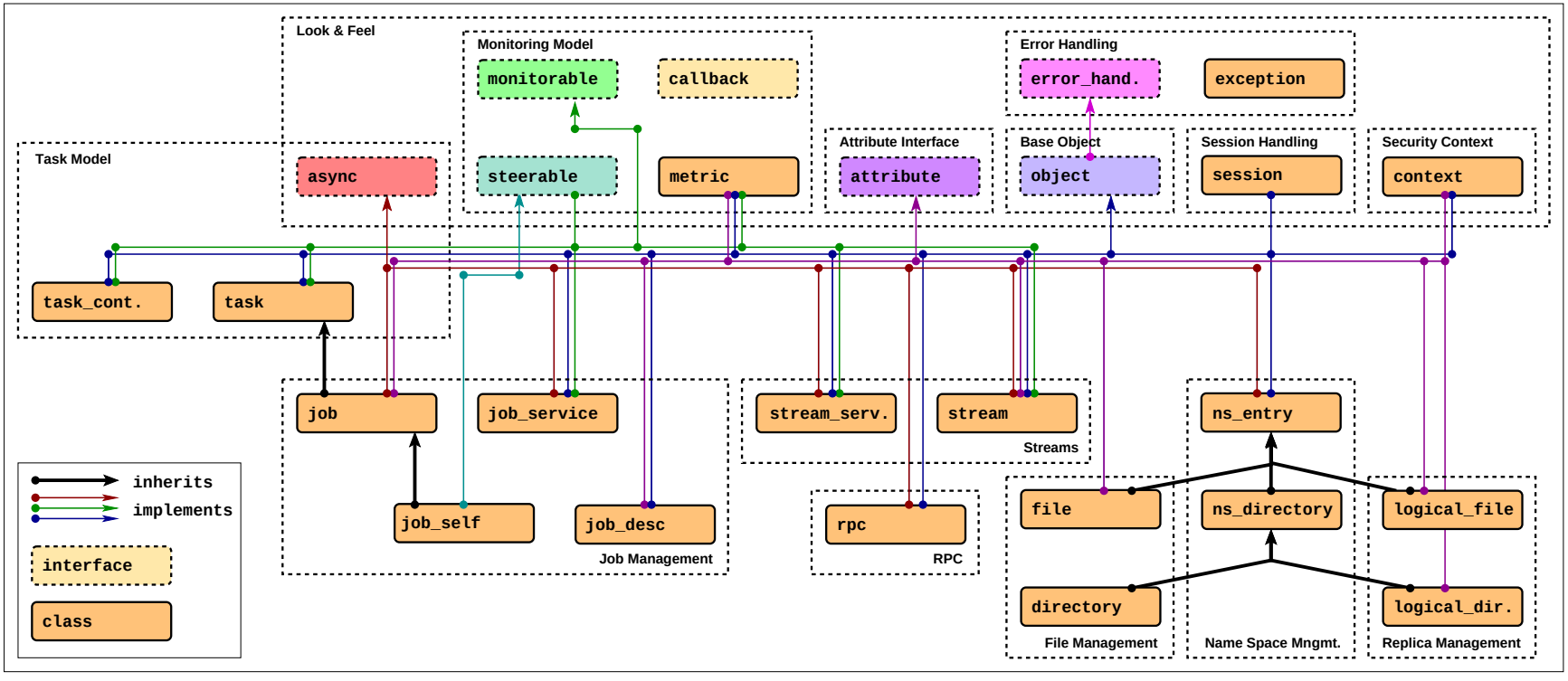
```
class my_cb : public saga::callback
{
  public:
    bool cb (saga::monitorable obj,
             saga::metric      m,
             saga::context     c)
    {
      cout << "Memory Usage: " << m.get_value () << endl;
      return (true);
    }
};

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


j.add_callback ("MemoryUsage", cb);
```

SAGA Examples: Monitoring

callbacks (cont.)

```
class my_cb : public saga::callback
{
  public:
    bool cb (saga::monitorable obj,
             saga::metric      m,
             saga::context     c)
    {
      cout << m.get_name ()  << " : " << m.get_value () << endl;
      return (true);
    }
};

list <string> metrics = j.list_metrics ();

while ( metrics.size () )
{
  j.add_callback (metrics.pop_front (), cb);
}
```

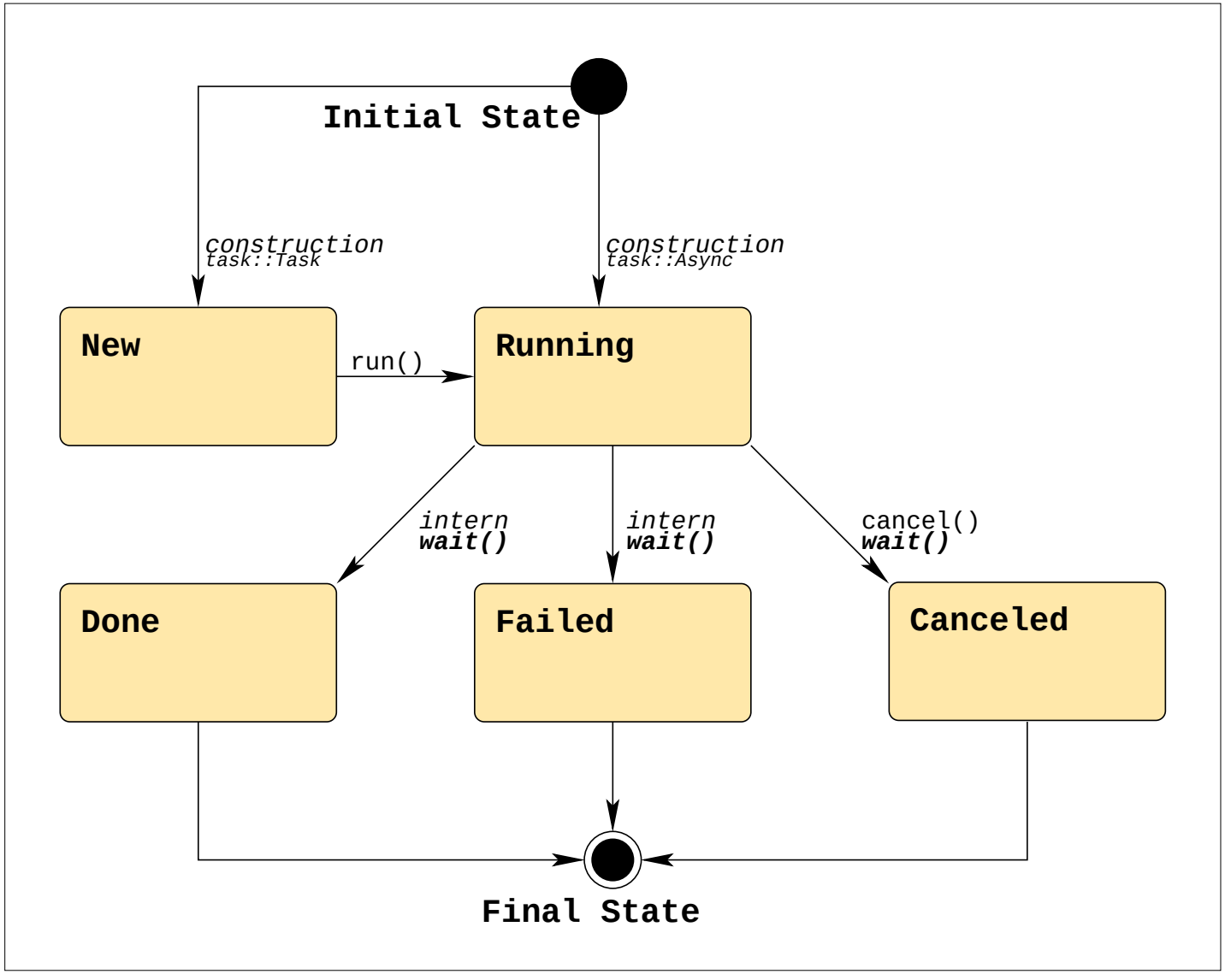
SAGA: Tasks


Look & Feel
Monitoring Model
monitorable
callback
Error Handling
error_hand.
exception
Task Model
async
steerable
metric
Attribute Interface
attribute
Base Object
object
Session Handling
session
Security Context
context
task_cont.
task
job
job_service
stream_serv.
stream
Streams
ns_entry
job_self
job_desc
Job Management
rpc
RPC
file
ns_directory
logical_file
directory
logical_dir.
File Management
Name Space Mngmt.
Replica Management
inherits
implements
interface
class

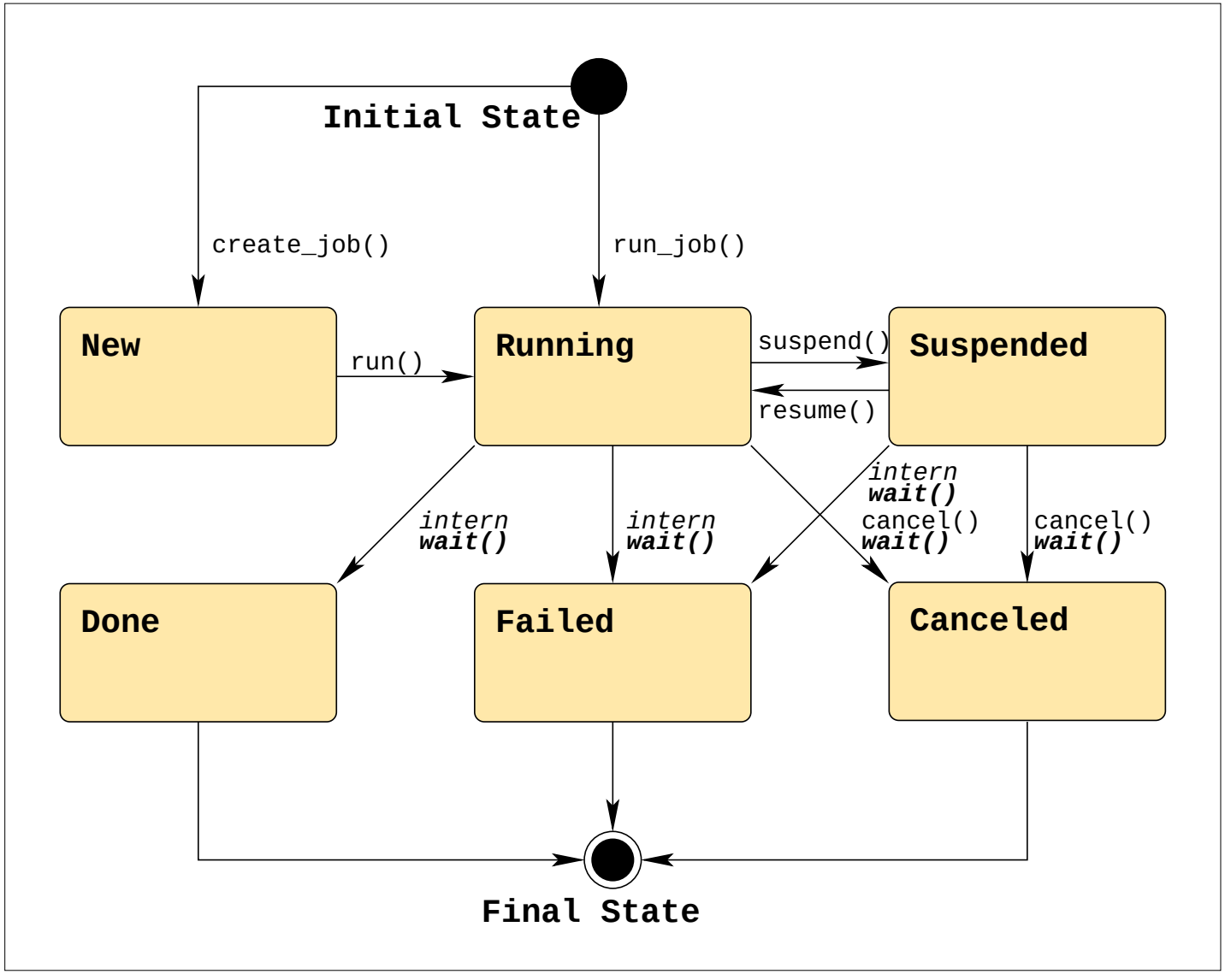
SAGA: Tasks

- asyncronous operations are a MUST in distributed systems, and Grids
- `saga::task` represents an syncronous operation (e.g. `file.copy ()`)
- `saga::task_container` manages multiple tasks
- tasks are stateful (similar to jobs)

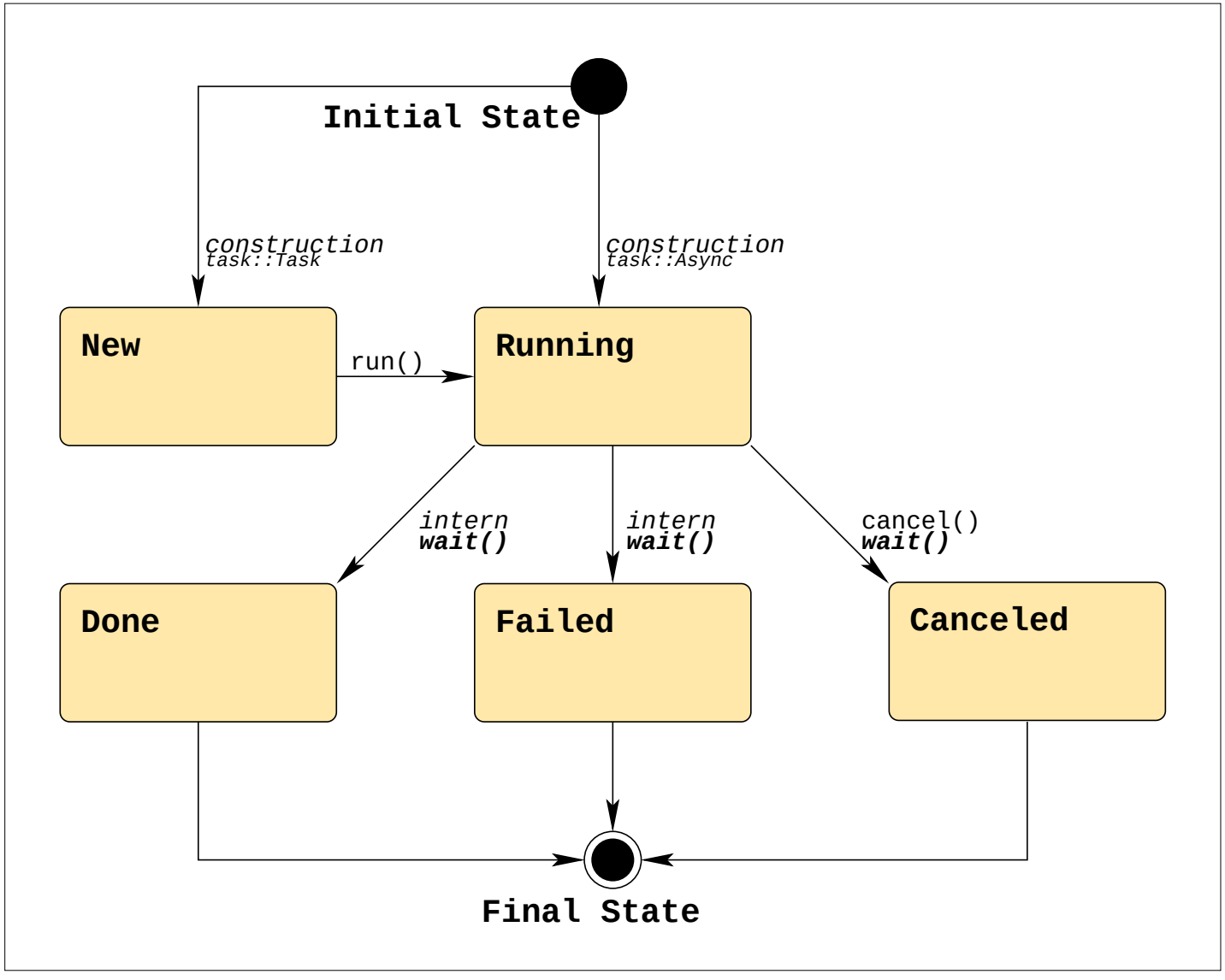
SAGA: Task States



SAGA: Job States



SAGA: Task States



SAGA: Tasks

- different versions for each method call: sync, async, task
- signature basically the same
- differ in state of task returned by that method

SAGA Examples: Tasks

tasks (i)

```
saga::file file ("gsiftp://remote.host.net/data/data.bin");

// normal, synchronous
file.copy ("data.bak");  // void

// async versions, never throw (use 'rethrow' on failure)
saga::task t1 = file.copy <saga::task::Sync>  ("data.bak.1");
saga::task t2 = file.copy <saga::task::Async> ("data.bak.2");
saga::task t3 = file.copy <saga::task::Task>  ("data.bak.3");

// t1: Done
// t2: Running
// t3: New
```

SAGA Examples: Tasks

tasks (ii)

```
saga::file file ("gsiftp://remote.host.net/data/data.bin");

// normal, synchronous
size_t s = file.get_size ();

// async versions
saga::task t1 = file.get_size <saga::task::Sync>  ();
saga::task t2 = file.get_size <saga::task::Async> ();
saga::task t3 = file.get_size <saga::task::Task>  ();

// get_result implies wait, and can throw!
size_t s1 = t1.get_result <size_t> ();
size_t s2 = t2.get_result <size_t> ();
size_t s3 = t3.get_result <size_t> ();
```

SAGA Examples: Tasks

tasks (iii)

```
t3.run ();

cout << t3.get_state () << endl;  // Running

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

// t1, t2, t3: Done (or Failed...)
```

SAGA Examples: Tasks

tasks container

```
saga::task_container tc;

tc.add (t1);
tc.add (t2);
tc.add (t3);

tc.run  ();

saga::task done_task = tc.wait (Any);

tc.wait (All);
```

SAGA Examples: Tasks

tasks and jobs

```
saga::task task = file.copy <saga::task::Asyn> ("b");
saga::job  job  = js.run_job ("remote.host.net", "/bin/date");

task.add_callback ("State", my_cb);
job.add_callback  ("State", my_cb);

saga::task_container tc;

tc.add (task);
tc.add (job);

tc.wait ();
```

- - -

SAGA planned extensions

- service discovery
- message based communication
- information service (Advert Service)
- resource discovery and management
- checkpoint & recovery (GridCPR)

SAGA v2: Messages

Messaging server

```
saga::sender snd ("tcp://localhost:1234");

saga::msg msg;

msg.set_size (100); // arbitrary size!
msg.set_data ("abcd...");

sc.send (msg);
```

Messaging client

```
char buf [13];
saga::receiver rec ("tcp://remote.host.net:1234");

// int size = rec.test ();

saga::msg = rec.receive (); // internal buffer allocation
```

SAGA v2: Messages

- messages are received intact or not at all
- implies protocol, but is silent about interop
- async zero copy implementation is possible

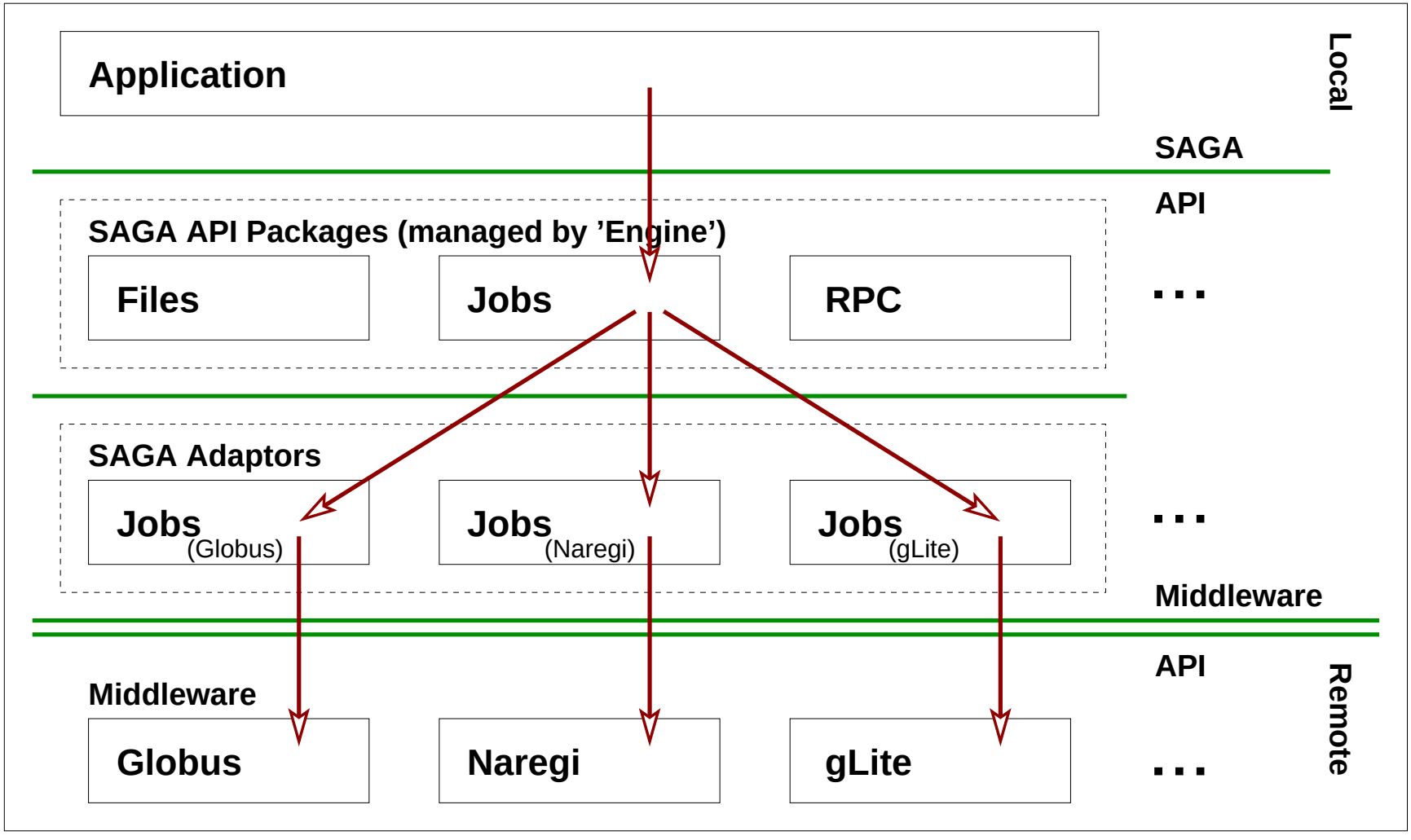
SAGA v2: CPR

- no examples yet, API in flux (serice spec in flux)
- allows to manage (find, move, stage, archive) checkpoints
- allows to trigger checkpointing of jobs
- probably name space based, with notification on CP creation

Questions about API?

Comments?

Implementation



Implementation Status

- C++
  - complete implementation by CCT/VU
  - in sync with spec, work in progress
  - other language bindings planned on top (C, Python, Perl, .Net)
- Java (out of sync with the spec)
  - partial implementation (jobs, files) by DEISA/EPCC
  - complete implementation by OMII-UK
  - possible complete implementation at VU
- MiddleWare Bindings
  - DEISA/EPCC binds to DEISA
  - OMII-UK binds to OMII stack, but is flexible
  - C++ adaptor based – GT4 and XtreemOS planned

Contact

`http://forge.ggf.org/sf/projects/saga-core-wg`

$\longrightarrow$ wiki, CVS details

Questions?

Backup Slides

Recap: Grid Applications

OpenGridForum

Types of Grid Applications

1. legacy applications
2. legacy distributed applications
3. Grid aware applications

Legacy Applications

- no access to application code
- virtualization of heterogenuity
- use cases: remote resource utilization, high throughput
- **favourite technique:** sandboxing (virtualization)
- no need for Grid APIs (application exists, non-mutable)

- *not a topic for this tutorial*

Legacy Distributed Applications

- aware of distribution (MPI, CORBA, ...)
- not aware of Grid properties (VO)
- usually not very dynamic or adaptive (bootstrapping!)
- use cases: scientific applications, bussiness applications
- **favourite technique:** emulation (GridMPI, etc.)
- no need for *new* Grid APIs (APIs non-mutable)

- *not a topic for this tutorial*

Grid Aware Applications

- aware of distribution, heterogeneity, VOs, elasticity etc.
- often dynamic and adaptive structure
- use cases: collaborative, adaptivite, auto-optimized, scalabile, ...
- **favourite technique:** depends on structure, middleware, ...
- need for Grid APIs (see Part 1)

- *topic for this tutorial :-)*

Grid APIs: Globus (pre-WS)

- low level API for the Globus Grid Middleware
- scope reflects Globus services:
  - GridFTP
  - GRAM
  - MDS
  - Replicas
- some low level API abstractions (xio, gss-assist)
- **CoG** provides higher level API abstraction for Globus (Java)

GridFTP Example

GridFTP: Connection Setup

```
globus_module_activate (GLOBUS_FTP_CLIENT_MODULE);

globus_ftp_client_handleattr_init        (&handle_attr);

globus_ftp_client_handle_init            (&handle, &handle_attr);
globus_ftp_client_handle_cache_url_state (&handle, server.c_str());
```

GridFTP Example (ii)

GridFTP: Get File Size

```
globus_ftp_client_operationattr_init     (&attr);
globus_ftp_client_operationattr_set_mode (&attr, ...);

globus_off_t    size    = GLOBUS_NULL;
globus_result_t success = globus_ftp_client_size
                                  (&handle,
                                   url.c_str(),
                                   &attr,
                                   &size,
                                   GLOBUS_NULL, // done_callback,
                                   GLOBUS_NULL);
if (success != GLOBUS_SUCCESS)
{ ... }
```

GridFTP

- API covers full scope of GridFTP protocol
- low level control over connection and operations
- allows syncronous and asyncronous calls (callbacks)

GRAM Example

GRAM: Job Submit

```
globus_gram_client_callback_allow   (callback_func,
                                     (void *) &Monitor,
                                     &callback_contact);

rc = globus_gram_client_job_request (rm_contact,
                                     job_description,
                                     job_state_mask,
                                     callback_contact,
                                     &job_contact);
```

GRAM

- API provides full scope of GRAM protocol
- low level control over operations
- syncronous and asyncronous calls
- job details encapsulated in job description (RSL)

GRAM Example (RSL)

GRAM: RSL example

```
+ ( &
    ( directory    = "/home/user/demo" )
    ( jobtype      =  mpi )
    ( executable   = "/home/user/demo/mpi-application" )
    ( maxWallTime  = "10" )
    ( count        = "8" )
    ( architecture = "i386" )
  )
  ( &
    ( directory    = "/home/user/demo" )
    ( jobtype      =  mpi )
    ( executable   = "/home/user/demo/mpi-application" )
    ( maxWallTime  = "10" )
    ( count        = "16" )
    ( architecture = "i386" )
    ( resourceManagerContact = "fs2.das2.nikhef.nl" )
)
```

GRAM - RSL

- GRAM comes with its own job / resource description language
- most middlewares invent their own languages
- requirements are interpreted in several places (resource broker, queue manager, . . .)

gLite Example

gLite: Job Submit

```
client.Delegate (delegID,
                 "https://cream-ce-01:8443/.../CREAMDelegation",
                 "/tmp/x509up_u202");
client.Register ("https://cream-ce-01:8443/.../CREAM",
                 "https://cream-ce-01:8443/.../CREAMDelegation",
                 delegID,
                 JobDescriptionBuffer,
                 "/tmp/x509up_u202",
                 uploadURL_and_jobID,
                 0, false);
client.Start    ("https://cream-ce-01:8443/.../CREAM",
                 uploadURL_and_jobID[1]);
```

gLite

- moves security details to API level
- in some sense, is a customized globus like environment
- shows its Globus foundations
- faithful to the web service paradigm (app-level WSDL)
  $\rightarrow$ API reflects gLite service interface
- JDL vs. RSL

CoG Example

CoG: Job Submit

```
String gramContact = "pitcairn.mcs.anl.gov:6722:...";
String rsl         = "&(executable=...)(...)(...)";

GramJob job = null;
try {
   job = new GramJob (rsl);
   Gram.request (gramContact,job);
}
catch (GramException e) {
   ...
}
```

CoG

- covers same scope as Globus API
- hides complexity and API evolution
- separates functional and non functional API parts

- new versions provide additional functionality (workflow, GUI, . . .) and cover non-globus middleware

GAT Example

GAT: Job Submit

```
ResourceBroker              rb ();
SoftwareDescription         sd (("location",  "/bin/ls")
                                ("arguments", "-l"));

HardwareDescription         hd (("memory.size",  1024.f)
                                ("disk.size",    10.f)
                                ("machine.type", "i686"));

Job job = broker.SubmitJob (JobDescription (sd, hd));
```

GAT

- tries to abstract Grid Middleware functionality
- tries to hide middleware details
- implementable on multiple middleware systems

- usability limited by scope of its use cases (historical reasons)