

The SAGA Task Model: Asynchronous Operations for Grid Applications

Andre Merzky, Thilo Kielmann

Vrije Universiteit, Amsterdam

<http://wiki.cct.lsu.edu/saga/space/start>

{merzky,kielmann}@cs.vu.nl



Simple API for Grid Applications (SAGA)

- **SAGA-RG** within GGF is working on an upcoming standard for a simple Grid API
- **80 / 20 rule:**
 - With 20% of the effort achieve 80% of the functionality
 - Strictly driven by application use cases
- API defines large part of a Grid programming model



What belongs to a Grid API?

- **Functionality, e.g. (as in SAGA today)**
 - **Jobs (submission and management)**
 - **Files (and logical/replicated files)**
 - **Streams**
 - **Later in SAGA:**
 - **Steering, monitoring, workflow, GridRPC, GridCPR**
- **Security**
- **Error handling**
- **Asynchronous operation (-> Tasks)**



SAGA Jobs

```
interface Job {  
  
    void getJobId          (out string          jobId);  
    void getJobState      (out JobState       state);  
    void getJobInfo       (out JobInfo        info);  
    void getJobDefinition (out JobDefinition jobDef);  
    void getJobExitStatus (out JobExitStatus exitStatus);  
    void suspend          ();  
    void resume           ();  
    void hold             ();  
    void release          ();  
    void checkpoint       ();  
    void migrate          (in JobDefinition  jobDef);  
    void terminate        ();  
    void signal           (in int           signum);  
  
}
```



SAGA JobService

```
interface JobService {  
    void submitJob (in JobDefinition jobDef,  
                   out Job job);  
    void runJob (in string host,  
                in string commandline,  
                out opaque stdin,  
                out opaque stdout,  
                out opaque stderr,  
                out Job job);  
    void list (out array<string,1> jobIdList);  
    void getJob (in string jobID,  
                out Job job);  
}
```



SAGA Files

```
class File {  
    void read      (in long      len_in,  
                  out string    buffer,  
                  out long      len_out );  
  
    void write     (in long      len_in,  
                  in string     buffer,  
                  out long      len_out );  
  
    void seek      (in long      offset,  
                  in SeekMode   whence,  
                  out long      position );  
  
    void readV     (inout array<ivec>  ivec);  
    void writeV    (inout array<ivec>  ivec);  
  
    ...  
}
```

(directories left out for brevity)



SAGA Replicated Files

```
class LogicalFile {  
  
    void addLocation      (in name                );  
    void removeLocation  (in name                );  
    void listLocations   (out array<string,1> names );  
    void replicate       (in name                );  
  
}
```

(directories left out for brevity)



SAGA Security

```
enum contextType {  
    X509                = 0,  
    MyProxy             = 1,  
    SSH                 = 2,  
    Kerberos            = 3,  
    UserPass            = 4  
};  
  
interface Context extends-all SAGA.Attribute {  
    constructor (in contextType type);  
    getType     (out contextType type);  
}
```

- **Every SAGA object gets a Context as parameter to its constructor.**



SAGA Error Handling

```
enum ExceptionCategory {
    LibraryRecoverableError,
    LibraryFatalError,
    BackEndRecoverableError,
    BackEndFatalError,
}

interface Exception extends sidl.SIDLException {
    getExceptionCategory (out ExceptionCategory category);
    getMessage           (out String           message);
}

interface ErrorHandler {
    hasError             (out boolean         state);
    getErrorObject      (out Exception      error);
}
```

Each SAGA object implements ErrorHandler.



SAGA Tasks

- Asynchronous operations
- Bulk (async.) operations
- Single-threaded implementation support

```
package Task {
```

```
enum State {
```

```
    Pending    = 0,
```

```
    Running    = 1,
```

```
    Finished   = 2,
```

```
    Cancelled  = 3
```

```
};
```

```
...
```



Tasks and Containers

```
interface Task {
    void run      ();
    void wait     (in double timeout,
                  out boolean finished);
    void cancel   ();
    void getState (out State state);
}

class TaskContainer {
    void addTask   (in Task task);
    void removeTask (in Task task);
    void run       ();
    void wait      (in double timeout,
                  out array<Task,1> finished);
    void cancel    ();
    void getStates (out array<State,1> states);
    void listTasks (out array<Task,1> tasks);
}
```



Instantiating Tasks is hard(er)

Model A:

```
DirTask dt = dir.createTask(); // new task classes
JobTask jt = job.createTask();

dt.copy(source, target); // same method signature as dir
jt.checkpoint();

dt.run();
jt.run();
```



Instantiating Tasks is hard(er)

Model B:

```
DirTaskFactory dtf = dir.createTaskFactory(); // new factory
JobTaskFactory jtf = job.createTaskFactory(); // classes

Task t1 = dtf.copy(source, target); // signature "similar"
Task t2 = jtf.checkpoint(); // but not 100% identical

t1.run();
t2.run();
```



Instantiating Tasks is hard(er)

Model C:

```
Task t1 = dir.task.copy(source, target);
```

```
Task t2 = job.task.checkpoint();
```

```
// task object within each object, similar/consistent signature
```

```
t1.run();
```

```
t2.run();
```



Instantiating Tasks is hard(er)

Model D:

```
Task t1 = dir.task_copy(source, target);
```

```
Task t2 = job.task_checkpoint();
```

```
// duplicate methods, explicit sync and async versions
```

```
t1.run();
```

```
t2.run();
```



Conclusions

- **Simple Grid APIs are both necessary and hard to define**
- **Have to combine**
 - **Functional properties**
 - **Non-functional: security, errors, asynchronicity**
- **Restrictions apply**
 - **Single-threaded**
 - **Bulk operations**
- **Did I mention components?**